Degree Final Project

Bachelor's degree in Aerospace Vehicle Engineering

Study: Visualization of spacecraft trajectories with NASA SPICE and Blender

REPORT

June 30, 2020

Author: Bernat Garreta Piñol

Directors: Manuel Soria Guerrero, Arnau Miró Jané

Call: 06/2020

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Superior d'Enginyeries Industrial, Aeroespacial i Audiovisual de Terrassa

Abstract

Visualizing spacecraft trajectories is interesting because of its compelling applications. From educational and public divulgation purposes, to mission planning and analysis, many fields can benefit from research in this topic. This project aims to provide with a tool to visualize such trajectories in a comprehensible three-dimensional manner. For this reason, a toolkit programmed in python and integrated into Blender and using the SPICE system has been developed.

On rare occasions SPICE and Blender have been integrated together. SPICE is a system created by NASA with the goal of assisting engineers and scientist in the planning and interpretation of space instruments observations, and in space exploration modeling and planning. It is a tool that analyzes data sets composed of navigation information to obtain precise observation geometry. Blender is a free and open-source 3D fully integrated 3D creation suite. It supports modeling, animation, rendering and video editing, among many other features.

The toolbox developed is able to recreate scenes by retrieving location, orientation and other data from SPICE, and applying this information to the animation. All space bodies are created as instances of different classes, which consist of planets, moons, stars and spacecrafts. The Blender scene is recreated using ancillary functions and the class functions of each object, which consist of creating the different objects in the scene, applying them their texture and animating the multiple bodies at each frame.

The toolkit makes use of the Blender API to access the program using its functions and classes. The in-built Blender renders Eevee and Cycles are used to obtain images and videos. The primitive mesh UV sphere is used to recreate the solar system bodies and stars, alongside materials with map textures. 3D models are used for the spacecrafts.

Some examples have been recreated to illustrate the capabilities of the developed toolkit. These consist of a recreation of the iconic image “Earth rising over the Moon’s Horizon”, an animation of the inner solar system during a whole year, a comparison of the constellations viewed from the Earth, two recreations of real images taken by Cassini one depicting three Saturn moons as they orbit over the rings and the other the moon Pan inside the Encke gap, an animation of the trajectory and attitude of Cassini during the insertion into Saturn orbit, and a recreation of an image taken by Voyager 1 showing an ongoing eruption on Io.

Overall, the toolkit is able to successfully recreate scenarios involving spacecrafts and their trajectories. It enables the user to comprehend the three-dimensional relative position of the bodies, and to recreate real images taken by spacecraft as well as simulate pictures that never happened. For these reasons, amongst its most important applications are education and dissemination.

Declaration of honour

I declare that,

- the work in this Degree Thesis is completely my own work,
- no part of this Degree Thesis is taken from other people's work without giving them credit,
- all references have been clearly cited,
- I'm authorised to make use of the research group related information I'm providing in this document.
- I understand that an infringement of this declaration leaves me subject to the foreseen disciplinary actions by The Universitat Politècnica de Catalunya - BarcelonaTECH.

Bernat Garreta Piñol		**June 30, 2020**
Student name	Signature	Date

Title of the Thesis : **Study: Visualization of spacecraft trajectories with NASA SPICE and Blender**

Contents

List of Figures

List of Tables

.

1 Introduction

The visualization of spacecraft trajectories is a complex topic that can be approached from many angles and points of view. This section presents how this work faces said objective, It explains the approach of the project to give an understanding of the reasoning behind the direction taken by this work, where it starts and where it is trying to go.

1.1 Aim

The aim of this project is to develop a code for the visualization of robotic exploration probes in the solar system using the NASA SPICE library on Blender with Python. Blender is an open-source 3D graphic software. It is a tool which is used for games, movies, 3D animation, 3D modeling, among other applications. There have been many 3D projects developed with this program. NASA SPICE is a ancillary information system used to compute geometric information. It is developed and maintained by NAIF, which is part of NASA, and is used in data analysis of planetary missions involving astrophysics, solar physics and earth science.

1.2 Scope

It is part of the scope of this project the development of a code that is capable of accurately representing spacecraft trajectories using Blender and NASA Spice Toolkit, alongside the development of all the necessary modules for the proper functioning of the code. It's also part of the scope to test the deployment in various computers and to study the applications of the code and possible future improvements.

It is not part of the scope of this project to develop accurate 3D models of the probes, to compute spacecraft trajectories with other tools than Spice nor to perform any code optimization.

1.3 Requirements

The project has the following requirements:

- The estimated dedication to the project is 600 hours.
- The documentation shall be written in English.
- The project shall be delivered before June 30.
- The code should be able to visualize what a certain spacecraft can observe.
- The code should be capable of helping visualize the trajectories of spacecraft, to show the public and for educational purposes.
- The code should work on any computer running Linux and Blender.
- The code should be easy and friendly to the user to a reasonable extent.

1.4 Usefulness of the project

There are many tools and codes in the market able to visualize spacecraft trajectories. Spice has been used for many projects, and Blender has already been used in space related work.

This project puts together these tools, something that has rarely been done before. The end product of this project is a tool to be able to visualize and explore spacecraft trajectories in a three dimensional environment. This can have potential applications in the fields of education, research or spacecraft mission planning.

2 State of the art

The SPICE system has already been used by many projects. It has been used in multiple space exploration missions, both from NASA and other space agencies [1]. A patent was created for a method and a computer system for processing spacecraft and planet trajectory data using SPICE, and although it is has expired, it shows the interest in SPICE applications [2]. A SPICE Module for the Satellite Orbit Analysis Program (SOAP) has been developed to precisely represent complex motion and maneuvers in an interactive, 3D animated environment [3]. Multiple organizations also create data sets and kernels to be used in SPICE. For example, the European Space Agency (ESA) generates the SPICE Kernel datasets for missions in all the active ESA Planetary Missions [4].

Blender has been used to create many 3D models of planets, satellites, and other space bodies. There are many tutorials available online on how to create planets, such as the YouTube video of "How to Make Earth in Blender (Cycles)" by Blender Guru [5], which is a very complete video explaining how to recreate the planet Earth with a high degree of complexity. The design of some materials created by this project is partially based on this work. A collection of Blender resources has been developed to streamline the creation of planetary objects [6].

SPICE and Blender have been very rarely used together. However, there is at least one instance that combined these two tools together. "WILL IT BLEND? GETTING SPICE-Y WITH DTMS AND PLANETARY VISUALIZATION" is a project that developed a tool to realistically animate terrain flyovers including spacecraft movement, using Blender for 3D model and animation, and SPICE to retrieve position information [7]. Their project is focused on recreating flyovers close to the surface, using digital terrain models to create planet and satellite surfaces with altitude and a high degree of exactitude. The Lunar Reconnaissance Orbiter Camera (LROC) team went on to use this tool to create a series of educational videos, which show impressive results [8]. However, this project does not share the same approach, as it is beyond the scope of this project to recreate geographical accidents. Although part of their videos could be recreated, it does not seem feasible that their results can be replicated within the scope of this work. For example, the rendering of one of their videos on a single high-powered workstation took over 1200 CPU hours, which corresponds with 50 days.

3 Concepts related to Space Exploration

The aim of this section is to provide the reader with a basic understanding of key concepts regarding space exploration. These concepts provide a context for the theoretical/mathematical development of the project. But first, the following terminology of space navigation used throughout this project has to be defined:

- Mission: used to refer to a whole operation, it is a general term that encompasses all aspects involved.
- Probe: General term that refers to any vehicle designed to fly in outer space. A mission usually consists of one single probe.
- Spacecraft: treated as synonym of probe and used interchangeably. S/C is a commonly used abbreviation of spacecraft.
- Orbit: it refers to the movement of a spacecraft that periodically rotates around a big body. Usually, spacecrafts enter into orbit of a planet. If a probe is designed to orbit some planet, it can be referred as an orbiter.
- Flyby: Operation in which the spacecraft flies close to a space body in order to change its trajectory using the gravity of that body, commonly referred as a gravity assist.

As a side note, most of the data of space missions is recorded during an orbit or a flyby, and as such most images are taken during these operations. Besides, most of the times to achieve the goal of the mission, such as getting to the targeted orbit or escape the solar system, a flyby is performed.

3.1 Time Systems

Time Systems are specified by a singular event (or reference epoch) at a certain instant, and every instant (or epoch) is measured by the time span to the reference epoch. Different time systems, apart from having different starting events, also compute the time in a different way, and as such, a conversion is needed to go from one system to another. Coordinated Universal Time (UTC) is defined as the civil time at Greenwich. The official time of each time zone is based on the UTC. Its unit of time is the atomic second, and it is expressed using the Gregorian calendar. UTC is based on the International Atomic Time (TAI), which is a purely atomic time system. However, the UTC follows the Earth rotation, which means that leap seconds are periodically added and subtracted to UTC because the Earth rotation varies through time, and these little changes are unpredictable. Barycentric Dynamical Time (TDB) is a relativistic time system intended for astronomical use and defined by the International Astronomical Union(IAU) [9]. J2000 is the epoch corresponding with TDB time 12:00:00 of January 1, 2000, and TDB is usually expressed as seconds past J2000. TDB is based on the Terrestrial Dynamic Time (TDT), which is in turn based on the TAI. These time systems relations are illustrated in the following figure 1:

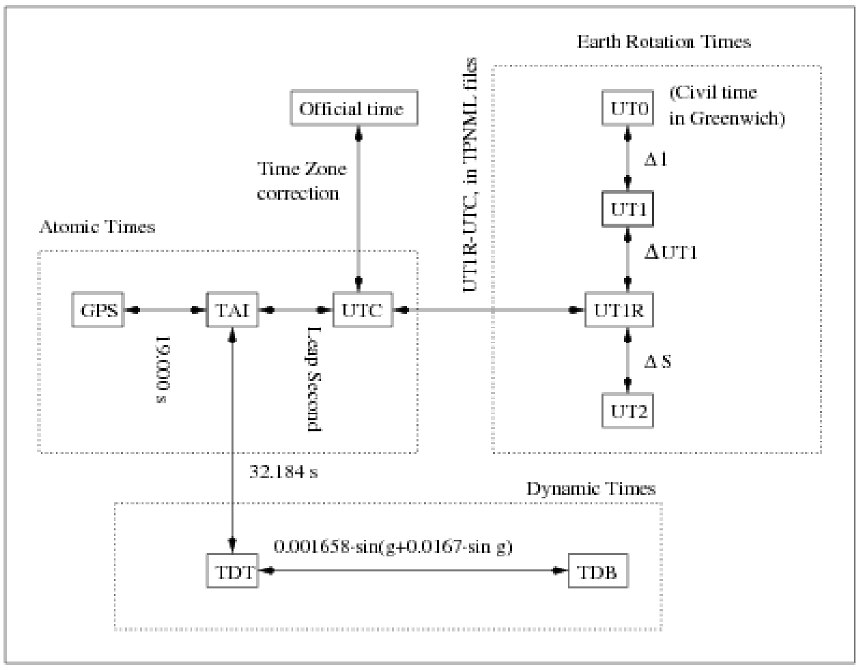


Figure 1: Time Systems Relations [10].

In this work, the relevant time systems are the UTC and the TDB. To go from UTC to TDB, the time has to be converted to TAI, then from TAI to TDT, and finally from TDT to TDB. These transformations, using the second as the time unit, are the following [10]:

$$\begin{aligned} UTC &= TAI - LS, \quad (1)\\ TDT &= TAI + 32.184, \quad (2)\\ TDB &= TDT + 0.001658\sin(g + 0.0167\sin g), \quad (3) \end{aligned}$$

where LS stands for Leap Seconds, and in the equation 3, the g is equal to:

$$g = \frac{2\pi(357.528^\circ + 35999.050^\circ T)}{360^\circ} \quad (4)$$

and T is equal centuries from J2000 in TDT. If we compute these transformations, the equation to transform from UTC to TDB is the following:

$$TDB = UTC + LS + 32.184 + 0.001658\sin(g + 0.0167\sin g). \quad (5)$$

3.2 Spacecraft clocks

Every spacecraft needs to track time so that it is able to determine at which instant actions are performed. These actions range from executing instrument commands to navigation orders. This is done using internal clocks, also called spacecraft clocks. There are many types of spacecraft clocks that range in a variety of complexity and accuracy. For example, the internal clock of the Ulysses spacecraft "was designed to increment its single field by one count every two seconds". On the other hand, the internal clocks of the Galileo and Magellan spacecrafts "were designed as four fields of increasing resolution" [11].

The time internally recorded by these clocks drifts relative to the other time systems. This drift can be calculated and predicted in order to improve the time conversion, however, even the most accurate spacecraft clocks end up losing precision. For example, the Deep Space Atomic Clock (DSAC) is a mercury-ion atomic clock that has very recently been developed by NASA. The DSAC is estimated to be "up to 50 times more stable than the atomic clocks on GPS satellites", however, Earth tests have shown that this extremely stable clock ends up losing one second every 10 million years [12].

3.3 Reference Frames

Reference Frames are defined by three orthogonal unit-length vectors that may vary in time and provide the base for the coordinate systems. There are two types: inertial and non-inertial. Inertial frames are non-rotating relative to the stars and their origin is not accelerating, meaning that inertial forces are negligible. In this work, the most relevant inertial reference frame is the International Celestial Reference Frame(ICRF) [13]. The ICRF is managed by the International Earth Rotation Service (IERS) and "is defined based on the radio positions of 212 extragalactic sources distributed over the entire sky", as shown in the figure 2.

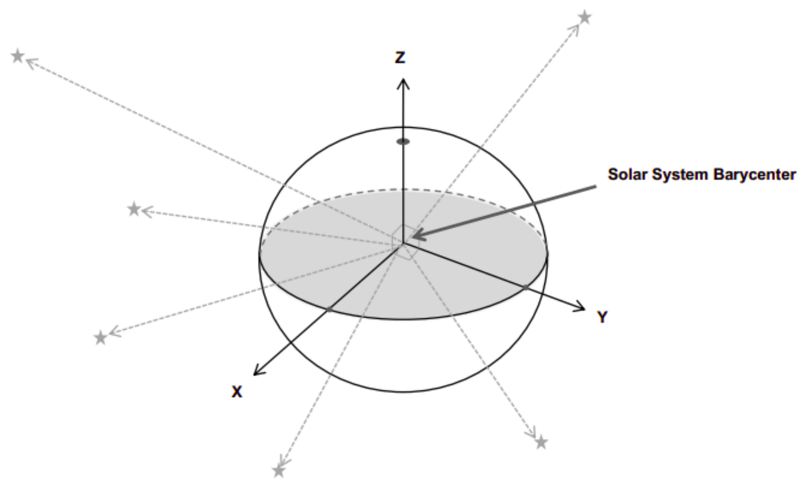


Figure 2: Representation of the ICRF [14].

The ICRF was made to be almost coincidental with the Earth-centred Equatorial Coordinate System (EME2000). This frame is also quasi-inertial and only differs from the ICRF by a rotation of less than 0.1 arc seconds [15]. The EME2000 is also known as J2000, however, the main difference lies in its definition [16]. The EME2000 frame defines the x-axis as "the intersection of the equatorial and ecliptic reference at J2000" and the z-axis as "the rotation axis of the Earth at J2000". J2000 is the J2000 epoch corresponding with TDB time 12:00:00 of January 1, 2000 as shown in the figure 3.

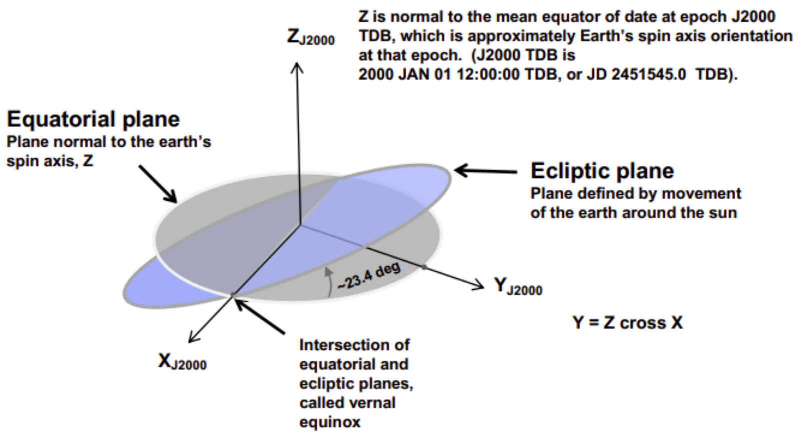

Z is normal to the mean equator of date at epoch J2000 TDB, which is approximately Earth's spin axis orientation at that epoch. (J2000 TDB is 2000 JAN 01 12:00:00 TDB, or JD 2451545.0 TDB).
Z_{J2000}
Equatorial plane
Plane normal to the earth's spin axis, Z
Ecliptic plane
Plane defined by movement of the earth around the sun
~23.4 deg
Y_{J2000}
Y = Z cross X
X_{J2000}
Intersection of equatorial and ecliptic planes, called vernal equinox

Figure 3: Representation of the EME2000/J2000 [14].

The most relevant non-inertial frames in this work are the body-fixed frames, the spacecraft frames and the instrument frames. Body-fixed frames are fixed to a solar system object, such as a planet, a satellite or the Sun. Spacecraft frames are fixed to the "bus" of an spacecraft, which is the center of the structural body of the spacecraft containing its payload and main subsystems, such as the electrical power or the communication subsystems. Instrument frames are fixed to an instrument of a spacecraft, with one axis pointing in the same direction as said instrument. To go from one frame to another, it is necessary to use a transformation matrix, which normally varies in time. The rotation inertial to body matrix(R_{IB}) that transforms from a inertial frame to a body-fixed frame is usually defined by three Euler angles, namely the right ascension(RA), the declination(DEC) and the prime meridian location(W). R_{IB} is obtained using the product of three basic rotations. First, a rotation($R_z(W)$) of W radians around the z axis:

$$R_z(W) = \begin{bmatrix} cos(W) & sin(W) & 0 \\ -sin(W) & cos(W) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{6}$$

Afterwards, a rotation ($R_x(\pi/2 - DEC)$) of $\pi/2 - DEC$ around the x axis:

$$R_x(\pi/2 - DEC) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & sin(DEC) & -cos(DEC) \\ 0 & cos(DEC) & sin(DEC) \end{bmatrix} \tag{7}$$

And finally, a rotation ($R_z(\pi/2 + RA)$) of $\pi/2 + RA$ around the z axis:

$$R_z(\pi/2 + RA) = \begin{bmatrix} -sin(RA) & cos(RA) & 0 \\ -cos(RA) & -sin(RA) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{8}$$

The rotation matrix(R_{IB}) is obtained with the product of these three basic matrices:

$$R_{IB} = R_z(W)R_x(\pi/2 - DEC)R_z(\pi/2 + RA) \tag{9}$$

R_{IB} transforms a vector(V) expressed in Cartesian coordinates in a inertial frame to a vector(V') expressed in a body-fixed frame, as shown in 10:

$$\mathbf{V}' = R_{IB}\mathbf{V}, \tag{10}$$

which can also be expressed as 11:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = R_{IB} \begin{bmatrix} x \\ y \\ z \end{bmatrix}. \qquad (11)$$

3.4 Aberration corrections

Because of the large distances and velocities in space exploration, there are non-negligible differences between the apparent location of an object and their actual location. It is also know as the geometric or uncorrected location. Aberration corrections need to be applied to the geometric vectors in order to obtain the apparent vectors. This term refers to all adjustment meant to correct the apparent position. These corrections are useful for accurately depicting instruments, in other words, what they actually see or where should they be pointed to. For this project, two aberration corrections are relevant: light time correction and stellar aberration correction [18].

The light time correction deals with the light time delay. Light travels in vacuum at a constant speed of $299,792,458\ m/s$, and for most purposes it can be considered instantaneous. However, due to the enormous distances between bodies that are dealt with in space exploration, the light time delay is not negligible. Therefore, to determine the apparent location and rotation of a body (target) located at R_B at a given time t from the perspective of a point (observer) located at R_A, the location of R_B needs to computed at $t - \Delta t$, where Δt is the time that takes the light to go from point B to A. As such, the light time corrected apparent location of B from A is given by a vector(r) that is determined as follows:

$$r(t) = R_B(t - \Delta t) - R_A(t), \qquad (12)$$

where the time delay Δt can be computed using the speed of light (c) and the distance that the light has to travel:

$$\Delta t = \frac{|r_{AB}(t)|}{c}. \qquad (13)$$

This two operations can be easily iterated to obtain an acceptable result, as the procedure converges rapidly [18]. A similar procedure is followed to obtain the apparent rotation. The result from these corrections is illustrated in the figure 4:

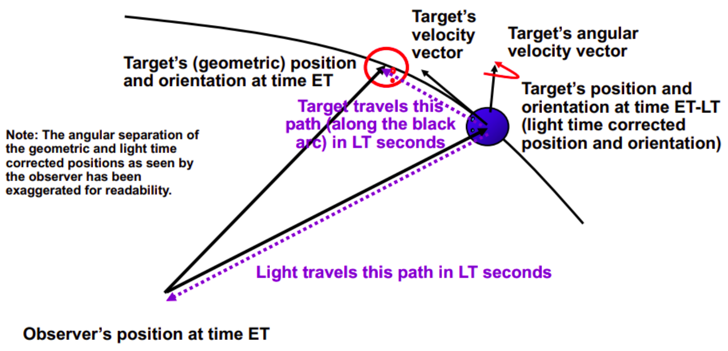

Target's velocity vector
Target's angular velocity vector
Target's (geometric) position and orientation at time ET
Target's position and orientation at time ET-LT (light time corrected position and orientation)
Target travels this path (along the black arc) in LT seconds
Note: The angular separation of the geometric and light time corrected positions as seen by the observer has been exaggerated for readability.
Light travels this path in LT seconds
Observer's position at time ET

Figure 4: Light Time Corrections [14].

Light time corrections are not enough, stellar aberration also needs to be corrected to obtain an accurate apparent location. Stellar aberration occurs when the observer is in motion. The observer velocity component orthogonal to the location of the target affects the direction at which the observer perceives the light that arrives from the target. Ignoring relativistic effects, which are small compared to Newtonian effects, the stellar aberration corrected location (r_p) ends up being computed as follows:

$$r_p = r + \Delta t \times v, \tag{14}$$

where r is the light time corrected apparent position, Δt the time it takes the light to reach the observer, and v the velocity of the observer relative to the solar system and orthogonal to the target location. The result of the light time and stellar aberration corrections is known as the proper apparent location, and is illustrated in the figure 5:

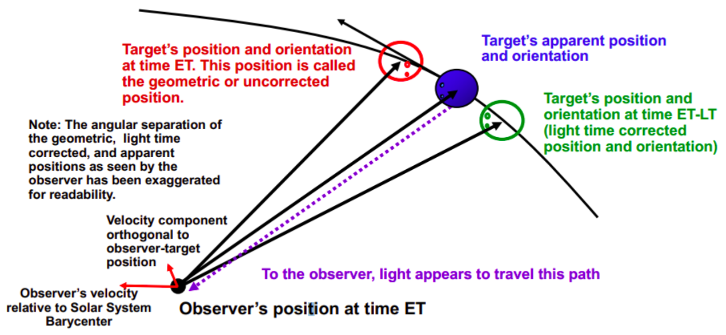

Target's position and orientation at time ET. This position is called the geometric or uncorrected position.
Target's apparent position and orientation
Target's position and orientation at time ET-LT (light time corrected position and orientation)
Note: The angular separation of the geometric, light time corrected, and apparent positions as seen by the observer has been exaggerated for readability.
Velocity component orthogonal to observer-target position
To the observer, light appears to travel this path
Observer's velocity relative to Solar System Barycenter
Observer's position at time ET

Figure 5: Stellar Aberration Correction [14].

Aside from the light time delay and stellar aberration, there are also other factors that affect the apparent location of an object, however, they have not been treated here.

4 SPICE System

The National Aeronautics and Space Administration(NASA) created the SPICE system in order to aid in mission engineering, science observation planning, and science data analysis [19]. This system was implemented by the Navigation and Ancillary Information Facility (NAIF) under the direction of NASA [20]. NAIF is a team dedicated to the development and deployment of SPICE, and it is responsible for producing high precision, clearly documented and readily used "ancillary information" required by space scientists and engineers [21]. The SPICE system begins with the production of ancillary data. This data is then read by the SPICE toolkit, which is designed to analyze and process it for its use. This toolkit has been used in most missions conducted by NASA since the Magellan mission in 1989 and by other space agencies such as the European Space Agency (ESA) or the Japanese Aerospace Exploration Agency (JAXA) [1].

SPICE toolkit was originally coded in FORTRAN, and was later ported to other programming languages by NAIF, among which was C (CSPICE) but not Python. Since Blender works with Python, it became necessary to use SpiceyPy, which is a CSPICE interface authored and maintained by Andrew Annex [22]. Although SpiceyPy has not been tested nor endorsed by NAIF, they provide links to and reference multiple times SpiceyPy in their official website. Moreover, NAIF states in their website that “SpiceyPy has been repeatedly vetted and tested both by users and extensive unit test run in a continuous integration service” [23].

4.1 Kernels

The data sets of ancillary data used by the SPICE toolkit are called kernels or kernel files [24]. There are different kinds of kernels, with each kind containing different types of data, and they come in two forms: binary and text kernels. The NAIF web page contains a large pool of reliable kernels obtained from mission data and scientific studies that enables the user to accomplish most tasks, however, kernels can also be created independently by any organization or user following the kernel architecture described in the NAIF web page.

The program must first load to memory the kernel in order to use its data with the SPICE toolkit. This is done with the function **furnsh**(***path***) [25], indicating the path of the kernel. The order at which kernels are loaded is important: kernels loaded last take precedence over those loaded before, so to avoid conflicting data between different kernels. At any point, the number of loaded kernels can not exceed 5300, among other restrictions. Therefore, kernels can also be unloaded to allow for more kernels to be loaded, although this is rarely needed.

The function **kinfo**(***file,typlen,srclen***) [26] is used to retrieve information of a loaded kernel. It returns the type of kernel, the source if it was not loaded with **furnsh**(***path***), and the handle if it is a binary kernel. If, for whatever reason, SpiceyPy is unable to locate the specified kernel, the exception *NotFoundError* is raised and the execution of the code is terminated. This can be avoided using the *no_found_check*() context manager, which makes it so the function **kinfo**(***file,typlen,srclen***) returns a fourth parameter that consists of a Boolean indicating whether it was possible to locate the kernel.

4.2 Naif Integer Codes

The SPICE system refers to objects, frames, and instruments by integer codes, also known as IDs. These IDs are unique to each object, frame or instrument, and each ID can have multiple names associated with it. Information about the integer codes and their associated names can be found at the NAIF website and inside each kernel containing relevant information[27]. The function **bodc2n**(***code, lenout=256***)[28] converts a SPICE ID to a corresponding name, while the function **bodn2c**(***name***)[29] does the opposite, converting a name to its corresponding ID.

4.3 Time Conversion and Time Formats

SPICE supports many different time systems, although only three are commonly used. These are the Ephemeris Time (ET), the Coordinated Universal Time (UTC), and the Spacecraft Clock Time (SCLK). In SPICE documentation, Ephemeris Time (ET) and Barycentric Dynamical Time (TDB[1]) are treated as synonyms [30]. For all purposes, ET and TDB can be used interchangeably.

ET is expressed in SPICE as seconds past J2000. This time system is used throughout SPICE as the independent time variable, since it is reliable and consistent. UTC needs to be transformed to ET in order to be used by SPICE. To do this, leap seconds need to be taken into account. To perform a conversion from UTC to ET in SPICE, a leap seconds kernel(LSK) needs to be loaded. This kernel contains all the leap seconds that have occurred and is regularly updated by NAIF. To perform an accurate conversion, this kernel needs to be up to date. An outdated LSK will result in time errors. The function **str2et**(***time***)[31] converts a string representing a time in TDB, TDT, or UTC to seconds past J2000.

Spacecraft Clock (SCLK) refers to the spacecraft internal time [32]. SCLK kernels contain the data necessary to perform conversions between SCLK and ET or UTC for a specific mission. SCLK kernels model the evolution of the time drift between SCLK and ET as a piecewise linear function and they are frequently updated. The function **sce2c**(***sc,et***)[33] converts, given the ID of a spacecraft, an epoch specified in ET to an encoded space clock value, representing ticks.

4.4 Reference Frames

SPICE provides the means to easily transform from one reference frame to another [34]. Inertial frames are built-in, meaning there is no need to load any kernel to obtain their attitude. It must be noted that the ICRF frame is also called J2000 by the SPICE documentation, not to be confused with the J2000 epoch. This is because SPICE considers that the EME2000 coincides exactly with the ICRF, and as explained in the section 3.3, EME2000 is also knwon as J2000. For backwards compatibility reasons, the SPICE toolkit only recognizes J2000 as a name for ICRF.

To obtain the attitude of non-inertial frames, which mainly comprise of body fix frames, an appropriate kernel needs to be loaded. Body fixed frames enable us to obtain the attitude of the body or object they are attached to relative to some other frame. In solar system bodies, the frame is fixed so that the x-axis corresponds with the intersection of the equatorial plane and the prime meridian, and the z-axis is "parallel to the mean axis of rotation and points north of the invariable plane of the solar system". In the case of spacecraft and instrument frames, the location of the axis for each case is usually defined in the relevant kernel.

For example, in the case of the Cassini probe, the orientation of the spacecraft and instrument frames is described in the Frames Kernel (FK). The Cassini frame is defined as follows: "The Z-axis emanates from the origin and is perpendicular to a plane generated by the mating surfaces

[1]The initials TDB come from the french *Temps Dynamique Barycentrique*

of the bus at bolt holes A, D, and H. The +Z-axis is on the propulsion module side of the interface. The X-axis emanates from the origin and is parallel to the line through the true centers of bolt holes A and H at the bus and the UEM upper shell structure assembly interface. The -X-axis points towards the Huygens probe. The Y-axis is mutually perpendicular to the X and Z axes, with the +Y axis oriented along the magnetometer boom" [35]. A graphical representation of the Cassini frame can be observed in figure 6, where $X_{S/C}$, $Y_{S/C}$ and $Z_{S/C}$ represent the three axis of said frame.

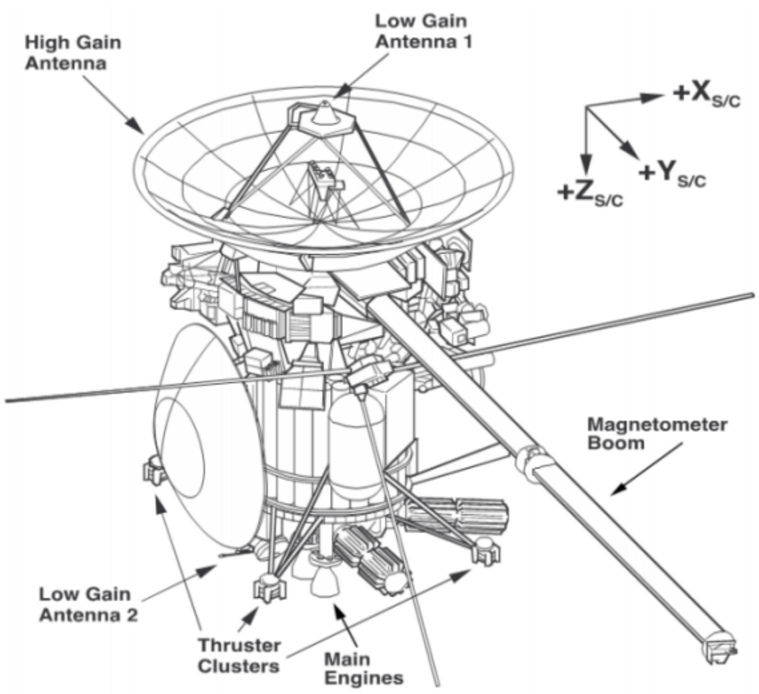


Figure 6: Cassini Configuration at Saturn Arrival [36].

For frames attached to solar system bodies such as satellites or planets Planetary Constants Kernels (PCK) are required [37]. They provide the attitude of multiple frames through time using orientation models. These orientation models use different variables, such as the right ascension, declination and prime meridian location. There are "tumbling" objects, that is, objects whose rotation cannot be accurately described using these models and as such they are not covered by a PCK. The function **pxform**(***fromstr, tostr, et***)[38] returns the transformation matrix for position vectors from one frame to another at a specified epoch, and the function **m2eul**(***r, axis3, axis2, axis1***)[39] transforms this matrix to the specified Euler angles in radians. The PCKs also provide shape information with a shape model. This shape model provides a three vector radii for each object. The function **bodfnd**(***body, item***)[40] returns whether values exist for an item of an object, and **bodvrd**(***bodynm, item, maxn***)[41] allows to retrieve the values for said item in case it exists.

Information of a spacecraft frame is contained in a C-kernel(CK) [42]. This type of kernel works in a very similar way to a PCK, however, one key difference is that CKs can have non-continuous data and gaps. Whereas with PCKs for every epoch covered in that kernel a transformation matrix can be obtained, with CKs this is not necessarily true. For this reason, a tolerance needs to be specified for SPICE to locate the closest data that falls inside this tolerance. The function

ckgp(***inst, sclkdp, tol, ref***) returns a transformation matrix given the ID of a spacecraft frame, an epoch and a tolerance in the corresponding SCLK, and a reference frame. It also returns an epoch in SCLK of the associated matrix, as it may not coincide with the desired epoch but still fall inside the tolerance. However, if multiple C-kernels are loaded containing data about the same frame, SPICE returns the data of the first kernel that falls inside the specified tolerance even if there is data close to the specified epoch in another kernel. As such, when working with multiple C-kernels, the tolerance should be as small as possible. Similarly to the function **kinfo**(***file,typlen,srclen***), the context manager *no_found_check*() is used to avoid an error if it is not possible to find the appropriate data inside the specified tolerance, and instead to obtain a Boolean indicating whether it was found. This matrix can be transformed to Euler angles using the aforementioned function **m2eul**(***r, axis3, axis2, axis1***).

Information of instrument frames are located in Frames Kernels. These kernels contain the attitude of each instrument relative to a spacecraft frame. These values are fixed and do not change in time. They can not provide a transformation on their own because they require the appropriated CK.

4.5 Positions and corrections

The Spacecraft and Planet Kernels(SPK) contain the position and velocity of all types of objects, be it solar system bodies or spacecrafts [44]. SPKs are formed by multiple segments, and each segment contains information regarding the position and velocity of some object (target) relative to a center of motion in a reference frame, and how they vary in the covered time. SPICE automatically chains different segments together when needed. For example, to obtain the location of the Huygens Probe relative to the Deep Space Station 14, represented on figure 7, the SPICE toolkit automatically finds and chains all necessary data contained in the SPKs, which consists of the vectors A to F.

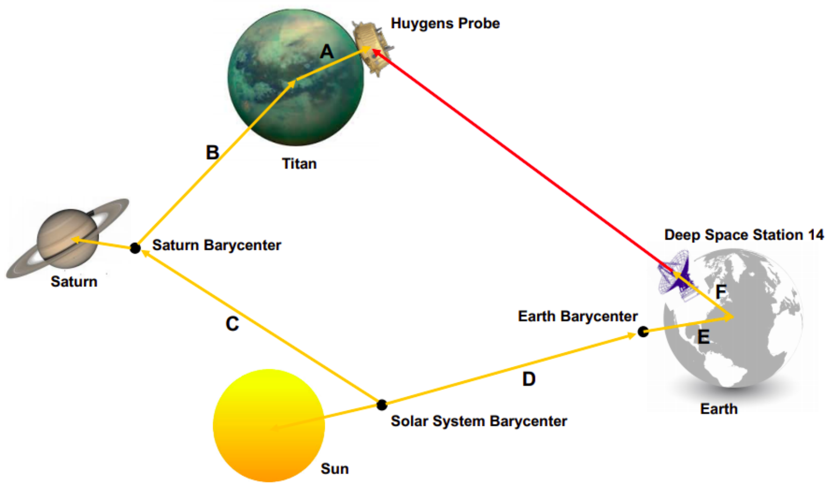


Figure 7: SPICE chaining process[45].

To obtain the data contained in a SPK, the function **spkezr**(***targ, et, ref, abcorr, obs***) is used [46]. It returns two three-dimensional vectors in Cartesian coordinates. The first vector is the position (in km) of the target body relative to the observer, and the second is the velocity relative to the observer (in km/s). The function also returns the time in seconds that the light needs to go from the target to the observer. The input parameters of this function are the ID or name of the target body, the epoch in seconds past J2000 TDB at the location of the observer, the frame of reference of the desired output, the corrections applied, and the ID or name of the observer body.

One of the input parameters is corrections applied. This refers to the aberration corrections applied to obtain the apparent location. There are mainly 3 options: 'None' does not apply any kind of correction, meaning that the function returns the geometric position, 'LT' applies a light time correction, and 'LT+S' applies a light time and a stellar aberration correction.

5 Blender

Blender is a free and open source fully integrated 3D creation suite that supports modeling, rigging, animation, simulation, rendering, compositing, motion tracking, video editing and game creation [47]. Although Blender was first released to the public by the dutch company Not a Number (NaN), nowadays it is developed and maintained by the Blender Foundation, an independent public benefit organization [48]. This project has used the Blender version 2.82, the most recent version when this project was developed. Blender 2.82 was released on 14 February, 2020 and it is freely available at the official website of Blender. Although new versions of the software may be eventually released that make parts of this project incompatible or outdated, all Blender versions that released are available to download [49]. All information about Blender 2.82 can be found at the Reference Manual, available at the official website [50].

5.1 Files

Blender saves all data in a Blender file, also know as a blend-file. Blend-files have a *.blend* termination, and each file "can contain multiple scenes and each scene can contain multiple objects. Objects can contain multiple materials which can contain many textures. It is also possible to create links between different objects, or share data between objects" [51]. A blend-file can also link data from other blend-files. Blender automatically creates backup versions whenever new data is saved. These files have a *.blend1* termination.

5.2 Scenes and objects

Scenes provide the base for objects in Blender. The properties of scenes relevant for this project are the active camera and the unit system. The active camera in a scene is the camera used for rendering. The unit system is only used in the user interface and for physics effects. For this reason, this project only focuses on the unit system in relevant situations.

Objects occupy the scene, and there are mainly three types relevant for this project: lights, cameras and meshes. Lights provide the lighting in the rendering of the scene and are invisible to the render. Cameras define what part of the scene is rendered and are also invisible. Cameras have many editable properties, such as type of lens, depth of field or aperture. Although one scene can have many cameras, the active camera at each frame will be the only one to be rendered.

Meshes are objects composed of polygonal faces. There are several primitive meshes, which consist of simple 3-D geometric models, such as a cube, a sphere or a circle, that are already provided by Blender. More complex shapes can also be created or imported by the user. Meshes can have various materials. Materials affect the reflection and/or emission of light by the different faces of the mesh. Materials use a system of nodes, where different types of nodes with assigned values are linked to other nodes, creating the resulting material.

All object types have many common properties, although only some are relevant for this project. Transform properties consist of the location, rotation and size or scale of the object in the scene. The location establishes the position of the object's origin, the rotation establishes the orientation, and the scale affects the size. Object relations establish parenting objects. All objects can become a child of a parent object, or become parents themselves. Any change in transform properties of the parent object, that is, movement, rotation or scaling, automatically affect the children. Although one parent can have multiple children, children can only have one parent.

5.3 Reference system and orientation

The scenes in Blender use a Cartesian coordinate system to define positions and rotations. This is called the global coordinates and the global axis. On the other hand, objects also have axis, which are called the local axis. These coordinates and axis are used in the transformation properties of the objects. To transform the position of an object, a new position has to be introduced. Each position in the scene is defined by the numerical values in the three global axis. By default, this is set to zero in each axis, which is the origin of the scene. To scale an object, new values in each axis for the relative scale are introduced. Scaling affects the scale of the object along its three local axis. By default, this value is set to one in each axis, which corresponds with the unaltered size of the object.

To rotate an object, a rotation mode has to be selected and then introduce the values, which can be in degrees or radians. There are three rotation modes, which are Euler, Quaternion and Axis Angle, although only Euler rotation is relevant for this project. In Euler rotation, a rotation order has to be selected. Blender uses an inverse order to name the axis order. For example, in the Euler rotation XYZ, the first rotation is around the Z axis, then around the Y axis, and finally around the X axis. Rotation uses the global axis, and by default it is set to zero.

5.4 Time management

Time in Blender is managed indirectly. Instead of working with seconds, Blender animates each frame. When the animation is rendered to obtain a video, the user establishes the frame rate, in frames per second. With this, a relation between frames and real time can be established. To render an animation, the user also has to define a start frame and a end frame. Although any frame can be animated, Blender will only render those frames that fall within the established frame range.

To animate in Blender, the user cycles through frames modifying the value of the current frame. Then, at the desired frame, the user modifies the properties of the scene. These properties that are modified usually consist of the location and rotation of objects, but there are almost no limitations. To save data of any property at a specific frame, a keyframe needs to be added. Keyframes mark the value of a property at a given frame. Afterwards, Blender interpolates the values of said properties for the frames that do not have keyframes. As such, it is not necessary to introduce a individual value for each frame. There are three types of interpolation, constant, linear and Bézier. Almost all properties defined by a numerical value can utilize keyframes.

5.5 Render engines

To render images or videos, Blender offers two in-built render engines, Cycles and Eevee. This project has used both Cycles and Eevee because they both have advantages and disadvantages for what this project aims to achieve. They share some features, although they both have features unique to each of them. Both Eevee and Cycles renders are defined by cameras, lights and materials.

5.5.1 The problem of scale

Before going into the Blender render engines, the problem of scale has to be discussed. The scale of distances poses a problem when trying to faithfully visualize space exploration. The root of the problem lies in the huge disparity between in object distances that have to be shown at the same time. This difference between many distances goes up to several orders of magnitude. For instance, the mean radius of Saturn is approximately 8 million times bigger than the total span of Cassini.

As an example, to recreate the Cassini mission to Jupiter, the bodies that at the very least should be visible at the same time are the Cassini probe, the planet Jupiter itself, its many satellites and rings, and the Sun, which is the main source of light. As can be seen in the table 1, the sizes of said objects are several orders of magnitude apart.

Object name	Approximate span	Order of magnitude
Cassini	$6.8m$	10^1m
Enceladus	$513.2km$	10^6m
Titan	$2,574.7km$	10^7m
Saturn	$58,232km$	10^8m
Sun	$696,342km$	10^9m

Table 1: Scale of objects.

Not only the different object sizes pose an issue, but also the distances between said objects, as can be seen in the table 2. Note that for the Cassini, the distance used is the shortest distance the orbiter was during the entry into Saturn orbit.

Object name	Distance to Saturn	Order of magnitude
Cassini	$20,000km$	10^8m
Enceladus	$237,948km$	10^9m
Titan	$1,221,870km$	$10^{10}m$
Sun	$1,433,530,000km$	$10^{13}m$

Table 2: Distance between objects.

This huge disparity in scale causes problems with the render engines of Blender. The reason is that they are not designed to deal with such different sizes and distances. Different solutions and workarounds have to be found for graphic problems that arise from this issue.

5.5.2 Eevee

Eevee is a physically based real-time render engine. Built using OpenGL, Eevee is focused on speed and interactivity. Eevee uses rasterization instead of computing each ray of light. Rasterization is a process that, using various algorithms, estimates how the light interacts with the objects in the scene and their materials. Eevee uses Physically Based Rendering principles, which means that it always provides a less accurate render than a ray-trace render engines such as Cycles, however, this loss in accuracy is traded for much higher speed of rendering relative to Cycles.

The use of rasterization in Eevee results in a large amount of limitations, and the treatment of indirect lighting is certainly relevant for this project. Eevee considers indirect light all light emitted or reflected objects. Indirect light is not treated the same as light that directly comes from light sources, called direct light. To account for indirect light, Blender has to “bake” the relevant parts of the scene. To bake a scene, Irradiance Volumes have to be added to the areas where indirect lighting is desired to be rendered. Their size can be increased, but then so has to do the resolution to obtain a decent render. This results in longer render times. After dealing with this, Eevee has to bake indirect lighting in those regions. This is an issue for animation, because Irradiance Volumes can not be moved through frames, and when Eevee bakes indirect lighting it does so for only one frame.

Clipping is a huge issue when using Eevee. The reason is that, because of the problem of scale, a huge clipping range is needed in order to observe the objects. This causes graphical problems with the Eevee render, as can be seen in the figure 8.

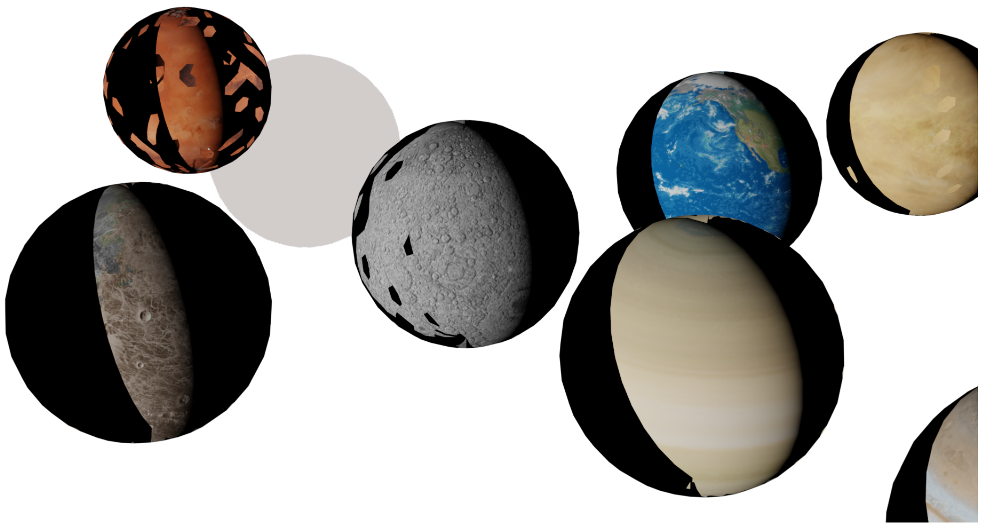


Figure 8: Graphical problems caused by clipping in Eevee.

One solution is to adjust the clipping range so that it is not too big. However, this has the downside of hiding objects that are either too close or too far. In the end, this is partially solved by using back face culling in the materials, which hides the back side of faces in the final render. Another workaround is to make the smaller objects, which mainly are the spacecrafts, some magnitudes larger than they actually are. In this way, the clipping range can be adjusted. The result can be seen in the figure 9.

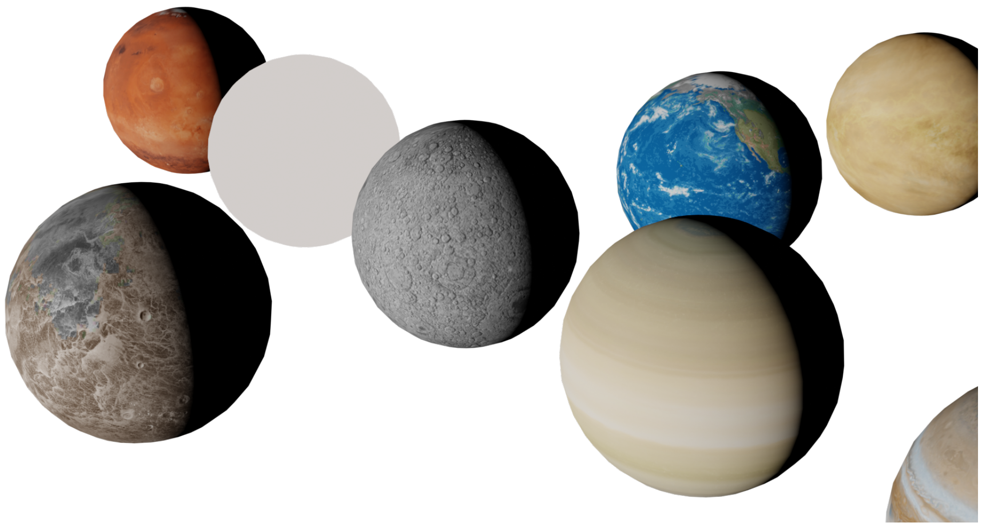


Figure 9: Graphical problems solved.

5.5.3 Cycles

Cycles is a physically based path tracer. This means that Cycles renders the scene computing all the light rays individually. Light rays are emitted from a light source or an emitting material. They are reflected up to a flexible maximum number of times by objects, and can be transmitted through transparent bodies. The result is a very realistic render, as the light closely mimics reality. However, it also has its downsides, mainly caused by the huge disparity in distances that are treated in this project, as explained in the subsection 5.5.1. To circumvent this problem and get a good result, a huge number of samples has to be set, causing the rendering to take a long time. In the end, Cycles trades rendering time for better image quality.

5.6 Python API

Blender has an embedded Python interpreter, that for this project is the Python version 3.7.4. This enables Blender to run Python scripts through the Text Editor and commands through the console. Moreover, Blender offers a Python Application Programming Interface (API), which is referred as *bpy* in Python. This API enables the scripts executed in Blender to perform all possible actions in Blender, such as creating a mesh, cycle through frames, add materials, and many others. All information regarding the Python API can be found in the documentation at the official [52].

6 Development of the toolkit

The development of the toolkit is a task that spanned months. After going through many versions and iterations, the final result is explained in the following sections. The functioning of the program is explained using pseudocode, and the code in its entirety can be accessed in a *github* repository[53].

The functioning of the toolkit is shown through a series of particular cases in section 7. Each case presents the solution to a specific goal and its results. The objective of each one is to explain how part of the toolkit works and show the results. The main reasons behind this approach is to make the explanation of the toolkit more understandable and to provide specific examples of what this toolkit can achieve.

6.1 Toolkit structure

The toolkit was developed with a user that has some basic programming knowledge in mind. It is therefore intended to be manipulated to suit the needs of the user, especially high order functions. Low order functions try to be reasonably flexible so that the minimum number of modifications are needed. The toolkit overall structure has been designed with the help of professor Miró.

The main folder containing all the necessary files is called *mView*, which stands for "Mission View". The user is only required to download this folder alongside Blender to be able to use the toolkit. The folder with the Blender materials created for this project is very heavy, and for this reason it has to be downloaded separately. The toolkit is under the "GNU General Public License v3.0".

The folder *materials* contains textures and models in the form of several Blender files that are used by the toolkit. There are two types, materials and spacecraft files. High res and low res files contain the different materials, created using shader nodes as seen on figure 10, that are used by the toolkit.

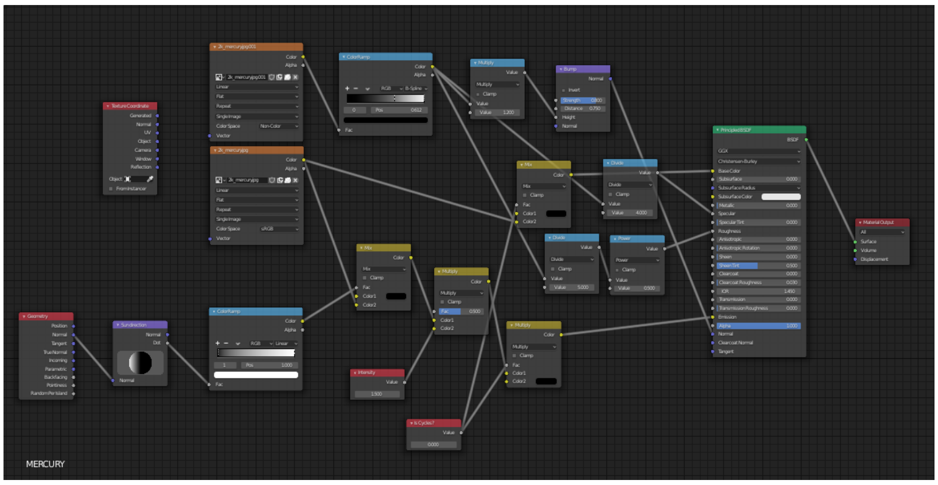


Figure 10: Overview of the MERCURY material in *LOW_RES.blend*.

The materials can get very complex, using many capabilities of Blender to try to achieve the maximum fidelity. This project does not delve deep into this aspect of the toolkit, and instead focuses in other aspects. It is pertinent to note that some work has been put behind the creation of materials for the toolkit. Although the different materials have been created specifically for this project, with the invaluable assistance of professor Miró, the textures used have been obtained from various open-source online sources, mostly based on observation and mission data. The materials are displayed in the Blender file using a set of simple spheres, as seen in figure 11.

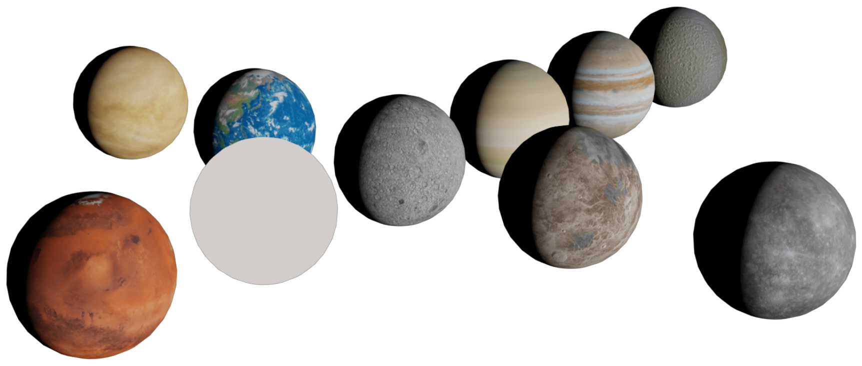


Figure 11: Overview of the *LOW_RES.blend* file.

The materials file contains both the accurately scaled mesh of the rings and the material used. The only rings created for this project are the Saturn rings, as seen in figure 12, although more rings could be created for other solar system bodies.

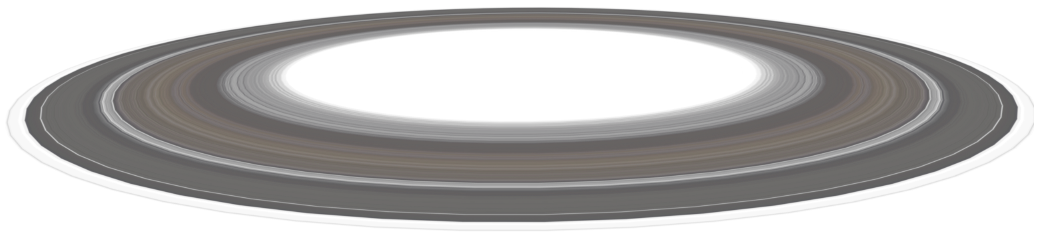


Figure 12: Saturn rings in the *LOW_RES.blend* file.

The spacecraft file contains the different 3D models of spacecrafts containing both the various meshes and materials. They have been obtained from the official website of NASA, that provides these resources for free. Figure 13 shows the Voyager model on the right, and the Cassini model on the left.

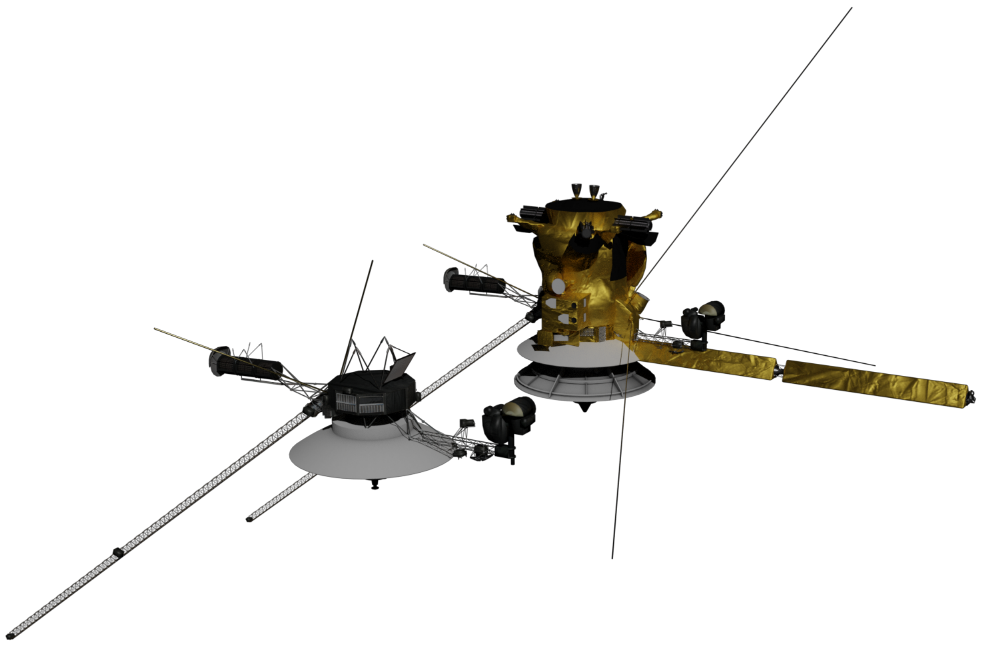


Figure 13: Cassini and Voyager models in the *SPACECRAFT.blend* file.

The folder *modules* contains two folders: *spiceypy*, which is the unaltered SpiceyPy package. In this way, the user is not required to separately download the SpiceyPy module. The other module is *Blenthon*, which is the folder containing all functions and classes developed for this project. As a curiosity, the name *Blenthon* is the joining of the words Blender and Python.

The folder *Blenthon* contains three code files, *BlenderAPI.py*, which contains the functions that interact with Blender, *SpiceAPI.py*, which contains the functions that interact with SPICE, and *Converter.py*, which contains the functions that deal with conversions of size, distance and light. Alongside them there is the *__init__.py* file. Out of all the functions contained in both *BlenderAPI.py* and *SpiceAPI.py*, only the ones specified to be imported in *__init__.py* are designed to be used in the main codes. The rest of them are used internally by the toolkit, and although the user could import and use them, this is not contemplated in the design. *Blenthon* also contains the submodules *celestialBodies*, which contains the classes used by celestial bodies, and *spacecraft*, which contains the classes used by spacecrafts. There is also a folder named *RESSLISTS*, the contents and function of which are explained in subsection 6.6.

Throughout the whole of *Blenthon* there are several sets of data in the form of json files. They are named *DataSet.json*, where *DataSet* is their respective name. The data contained in each of them is explained in detail in the subsections from 6.5 to 6.10. The data sets are located in the same directory of the code that uses them. Alongside them, there are the Python codes that created them, called *CreateDataSet.py*. Although they are not used by the toolkit, the reason they are inside the toolkit is so that the user can easily modify them to create new data sets. It is not necessary to use these codes to create or modify the data sets. The toolkit works correctly with the data sets as long as they follow the same structure.

6.2 Initial remarks

In order to successfully run the toolbox, two conditions must be met:

- First of all, the current working directory must be the folder *mView*, so that the many relative paths used by the toolkit work correctly. Using the console, changing the directory to the *mView* path, and then starting Blender, achieves this. Afterwards, the user only has to open through the Blender text editor the relevant main code and execute it. It is also possible to change the current working directory with other means, such as with the function *os.chdir*().

- The second condition is that the path to the *mView* folder is designated in Blender as the script directory. This enables Blender to import the scripts or codes used, and can be done by manually accessing the Blender Preferences or by designating the file path using the Python console. Then, Blender needs to be restarted so that this goes into effect. As a side note, the name of the folder *modules* is not inconsequential, as a script executed in Blender will only import scripts from a folder specifically named *modules*.

6.3 Blender functions

This subsection offers a list of functions from the file *BlenderAPI.py* that are intended to be used in the main code. The rest of the functions contained in said file are only internally used by the toolkit, although a user could modify it and use these functions even if this is not intended by their design.

- **clean**() deletes all the objects, meshes, materials, textures and images from the scene, in addition to setting the start frame to 0 and the end frame to 250, which is considered the default in Blender.

- **set_frames**(***frames***) sets the end frame to the specified number and the start frame to 1.

- **to_frame**(***frame***) moves the frame of the scene to the one indicated.

- **set_background**() sets the background of the scene as pitch black and renames it to *Universe*.

- **add_camera**(***name,location,rotation,lens=50.,clip_end=1000.,clip_start=1.e-3***) adds a camera to the scene with the indicated properties.

- **move_camera**(***name,aim,center,distance***) moves the camera designated as *name* so that it is located on the line that goes from the position of the body *aim* to that of *center*. The camera is situated *distance* units apart from the *center*. The function also rotates the camera so that it is pointing towards *aim*.

- **finish**(***cstart=None,cend=None,show_name=False***) sets the specified clip range to the Blender 3D view, and if indicated, makes every object show their name in the viewport. Finally, the function packs all the files used so that in any case no other file is required apart from the blend file to obtain the scene.

- **subdivide_object**(***name,levels=2,render_levels=3***) increases the polygon count of the object designated by *name*, therefore improving its quality on both the 3D view and on the render.

6.4 SPICE functions

This subsection offers a list of functions from the file *SpiceAPI.py* that are intended to be used in the main code. Similarly to the previous subsections, other functions only used internally by the toolkit could be used by the user but it is not intended in their design.

- **init_spice**(***mission=None***) loads all the required kernels into memory. This is explained in detail in the subsection 6.5.
- **timeperiod**(***start,end,frames***) returns a list of epochs. The number of epochs is determined by *frames*, and they are equally distributed from the instant *start* to the instant *end*, which both of them consist of a string in one of the many formats recognized by SPICE. This is normally expressed in UTC but it is not required.
- **get_name**(***id***) returns the SPICE name of an object given its associated ID.
- **get_tolerance**(***name,epoch1,epoch2***) calculates the tolerance for a certain spacecraft designated by *name* as the difference between the spacecraft clock time at two different epochs.
- **findAllInstances**(***spacecraft,planet,***[**...**]***,path=RESSLISTS_DEFPATH***) returns a list of epochs that correspond with a set of images taken by a spacecraft that meet certain conditions. This is explained in more detail in subsection 6.6.
- **epoch_to_utc**(***epoch,format='C',decimals=0***) transforms an epoch to a string representing an instant in UTC. The default behaviour uses a calendar format with no decimals, but this can be modified.

6.5 Kernel management

Kernel management is only performed by the function *init_spice*(*mission*), which is represented in algorithm 1. This functioning is based on an earlier MATLAB function created and provided by professor Soria. In a first instance, *init_spice* creates a folder named *kernels* in the *mView* directory if it does not exist yet. Inside this folder, the kernels used in SPICE are located. The *kernels* folder is organized in sub-folders that are also automatically created by this function. To know which kernels to load, the toolkit reads the *MetaKR.json* data set, which is created by the *CreateMetaKR.py* code. This file contains the name and directory of each kernel, and the link from where they can be downloaded. For each kernel, the function tries to locate them. If they do not exist, they are downloaded to the specified directory using the link provided. Afterwards, they are loaded into memory. This process is very useful for the user, the reason being that it does not require any input from them. The user only needs to run the function and the toolkit automatically does the work of finding all the required kernels for them. It is also useful when the work of multiple people needs to be shared, because the toolkit will always obtain the required kernels.

The optional argument of this function is *mission*, which, if needed, specifies the kernels of which mission need to be loaded. In a similar way as before, the toolkit automatically creates the required directories, downloads the required kernels, and loads them into memory. To know which kernels to load, the toolkit reads the *MetaMission.json* data set, which is created by the *CreateMetaMission.py* code. This file contains a dictionary with all the relevant kernels of each mission. Since the kernels usually are heavy files, it is useful that only the kernels relevant to the current mission are downloaded and loaded into memory. Note that in this first case no mission needs to be loaded to obtain the result.

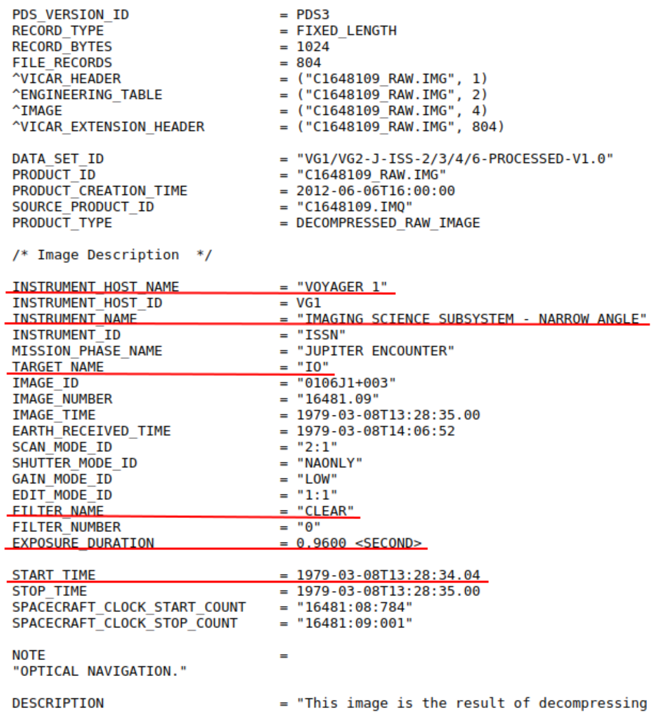
The following algorithm is a simplified representation of the actual code:

Algorithm 1: Representation of **init_spice**(***mission***).

```
Result: All necessary kernels are loaded into memory
if kernel directories not exist then
    create kernel directories
for kernel in MetaKR do
    if kernel not exist then
        download kernel
    furnsh kernel
if mission then
    if mission kernel directories not exist then
        create mission kernel directories
    for kernel in MetaMission do
        if kernel not exist then
            download kernel
        furnsh kernel
```

6.6 Mission meta data

The images taken by every NASA mission are available in different collections at The Cartography and Imaging Sciences Discipline Node [54]. These collections offer all the raw images taken of each mission alongside a text label. As seen on figure 14, this text label contains different types of data of the associated image, such as the instrument, target or filter. All of this mission meta data is compiled into the folder named *RESSLISTS* by professor Soria. *RESSLISTS* contains a folder for each mission, which currently consist of Voyager 1 and 2, and Cassini. These are further subdivided into a folder for each encounter or planet where there was a mission flyby or orbit. Inside a folder named *main_list* there are a set of text files, one for each image property, which consist of the image name, host, instrument, target, filter, exposure duration, start time and volume. Each line of text of each file corresponds with a single image. However, the different images are not ordered chronologically.



```
PDS_VERSION_ID                  = PDS3
RECORD_TYPE                     = FIXED_LENGTH
RECORD_BYTES                    = 1024
FILE_RECORDS                    = 804
^VICAR_HEADER                   = ("C1648109_RAW.IMG", 1)
^ENGINEERING_TABLE              = ("C1648109_RAW.IMG", 2)
^IMAGE                          = ("C1648109_RAW.IMG", 4)
^VICAR_EXTENSION_HEADER         = ("C1648109_RAW.IMG", 804)

DATA_SET_ID                     = "VG1/VG2-J-ISS-2/3/4/6-PROCESSED-V1.0"
PRODUCT_ID                      = "C1648109_RAW.IMG"
PRODUCT_CREATION_TIME           = 2012-06-06T16:00:00
SOURCE_PRODUCT_ID               = "C1648109.IMQ"
PRODUCT_TYPE                    = DECOMPRESSED_RAW_IMAGE

/* Image Description  */

INSTRUMENT_HOST_NAME            = "VOYAGER 1"
INSTRUMENT_HOST_ID              = VG1
INSTRUMENT_NAME                 = "IMAGING SCIENCE SUBSYSTEM - NARROW ANGLE"
INSTRUMENT_ID                   = "ISSN"
MISSION_PHASE_NAME              = "JUPITER ENCOUNTER"
TARGET_NAME                     = "IO"
IMAGE_ID                        = "0106J1+003"
IMAGE_NUMBER                    = "16481.09"
IMAGE_TIME                      = 1979-03-08T13:28:35.00
EARTH_RECEIVED_TIME             = 1979-03-08T14:06:52
SCAN_MODE_ID                    = "2:1"
SHUTTER_MODE_ID                 = "NAONLY"
GAIN_MODE_ID                    = "LOW"
EDIT_MODE_ID                    = "1:1"
FILTER_NAME                     = "CLEAR"
FILTER_NUMBER                   = "0"
EXPOSURE_DURATION               = 0.9600 <SECOND>

START_TIME                      = 1979-03-08T13:28:34.04
STOP_TIME                       = 1979-03-08T13:28:35.00
SPACECRAFT_CLOCK_START_COUNT    = "16481:08:784"
SPACECRAFT_CLOCK_STOP_COUNT     = "16481:09:001"

NOTE                            =
"OPTICAL NAVIGATION."

DESCRIPTION                     = "This image is the result of decompressing
```

Figure 14: Example of an image text label.

The treatment of *RESSLISTS* by the toolkit is accomplished with two functions located in the *SpiceAPI.py*. The first one, **getAllLists**(***path,mission***), is only used internally by the toolkit and is based on the MATLAB code *getAllLists.m* designed by professor Soria and available in the *RESSLISTS* folder. This function, given a path to the appropriate folder, reads the data for the specified mission. It also fixes the different errors and discrepancies of the data set that have been found. If the start time of an image is unknown, designed in the data as *UNK*, that image is discarded. The function returns a dictionary with the data of all images.

The second function is **findAllInstances**(***spacecraft, planet, seconds=None, host=None, target=None, filter=None, instrument=None, from_epoch=None, to_epoch=None, max=None, path=RESSLISTS_DEFPATH***), which is intended to be used in the main code. First of all, this function retrieves all the meta data of the specified *spacecraft* and *planet* or encounter with **getAllLists**(***path,mission***). The path to *RESSLISTS* is set by default, but should be changed if the folder is moved. The function cycles through the data of each image to check if they fit the criteria. The criteria is decided with the input attributes. If an attribute is left as *None*, then it is ignored by the function. Otherwise, for *seconds*, *host*, *target*, *filter* and *instrument*, the toolkit checks if they coincide with the meta data of each image. It also checks for each image if it was taken after the time specified by *from_epoch* and before *to_epoch*. Once the function has cycled through each of the images taken or it has found that a total of *max* images fit the criteria, the selected images are sorted chronologically. Finally, a list with the epochs of each image that fits the criteria is returned alongside a list with their name and volume, so that the original raw images can be easily found.

6.7 Converters

The converters are a set of classes defined in *Converter.py* that deal with unit conversions. There are three types of converters that are build on the same parent class, which comprise of size, distance and light converters. The converters have a dictionary of conversion factors for each type, such as the mean distance of Earth to the Sun or an arbitrary standard value. The converters are initialized by specifying the conversion factor, which can then be modified in the code. The toolkit uses the converters to transform physical values to be used in Blender. It does so by dividing the values by the specified factor. This can be a little bit confusing because it means that a bigger conversion factor results in smaller values in Blender. For example, if the size converter factor is decreased by a factor of ten, the resulting objects have a size ten times bigger. One object does not necessarily have to have the same converter for size and distance, which can be used to make objects bigger while maintaining the distances unaltered. Also, different objects can have different converters, which can be used to adapt the size of each object individually. If the distance and size converters in a scene have the same conversion for each object, the resulting scene is in real scale.

6.8 Lighting

The lighting of the scene is heavily affected by the render engine used in Blender. As such, the toolkit has to work differently depending on which render engine is intended to be used. The property **iscycles** is used to denote whether Cycles or Eevee is the render engine used.

If **iscycles** is true, then the toolkit considers that Cycles is the render engine used. In this case, the lighting of the scene is mainly provided by the Sun. The material of the Sun emits a powerful light that illuminates every object in the scene. The material of each object reflects the incoming Sun light. This light should be modified by the user to obtain the desired result. To a much lesser degree, the stars of the sky also provide some lighting, but this should be negligible. To achieve their goals, the user is expected to modify the render properties such as light path bounces, reflections, number of samples, and denoising, among others.

If on the other hand **iscycles** is false, then the toolkit considers that Eevee is the render engine used. As explained in the subsection 5.5, using indirect light in Eevee is not really viable. For this reason, and for other lighting problems that occur when Eevee tries to render extreme distance and size scales, a new different method of lighting is necessary.

Instead of having one main source of light in the Sun that is reflected by every object, the approach of Eevee lighting is that each object emits their own light. Since no indirect lighting is present, the different objects do not illuminate each other. A major drawback of this method is that, since each object emits their own light, shadows can not be recreated. This is a sacrifice in realism that is necessary for this method of lighting and is not easy to circumvent.

Although shadows are not be recreated, a method has been developed to recreate the night-day cycle of planets and satellites. All the materials used in these objects have a similar node structure, so that only the side of the ellipsoid that is facing the Sun is emitting light. The other side is in pitch black. This is accomplished with the *'Sun direction'* node. The function **update_light**(***name***), which is internally used by the toolkit, automatically updates this node with the position of the Sun at the current frame, as seen on algorithm 2.

Algorithm 2: Representation of **update_light**(*name*).

```
Result: Eevee lighting is updated for a body
active_object = object[name] for node in active_object.material.nodes do
    if node == "Sun direction" then
        node.value = Sun.location-active_object.location;
        node.keyframe_insert()
```

Finally, all the different materials have the *'Is Cycles?'* node. If the value of this node is one, then the material only reflects light and there is no emission. If the value is zero, then the material emits light according to the method previously described.

6.9 Object classes

All objects in the toolkit, from spacecrafts to celestial bodies such as planets or stars, are created as instances of predefined classes. After initializing them, the main code interacts with them using the defined class functions. All of the object classes except a minor exception have the class functions **build**, which creates the object in Blender, **set_material**, which adds the material to the object, and **to_epoch**, which moves the object to the specified epoch. Although they share the nomenclature and overall form to maintain consistency, each class function differs from the others in order to adapt them to the requirements of their specific class.

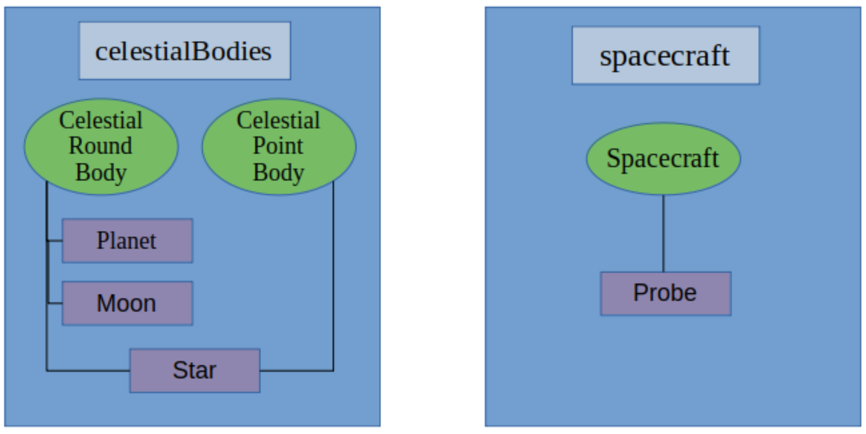


Figure 15: Diagram of Object Classes.

As seen on figure 15, the object classes are divided in two main groups, based on whether they are located on the *celestialBodies.py* or on the *spacecraft.py* file. The celestial bodies comprise of two base classes, celestial round and point body. These classes are in turn inherited by the classes Planet, Moon and Star. Planet and Moon objects always are round bodies, which means that they are represented in Blender as an ellipsoid. On the other hand, a Star object can be a round body, which should only be the Sun, or a celestial point body, which is used to represent the stars of the sky as a point of light. For spacecrafts, it is very simple, the class Probe inherits the class Spacecraft. Although it was not strictly necessary to have this structure for spacecrafts, it was decided to follow the same structure as the celestial bodies to accommodate future implementations.

6.9.1 Star class

The instances of the Star class are initialized with the following attributes:

- **name** is a string identifying the object. It is used as the name for the object in Blender and to retrieve the appropriate data, therefore it can not be an arbitrary name.
- **observer** is a string specifying which is the observer point or body. The toolkit considers that the observer corresponds with the point of reference or origin in Blender.
- **as_point** is a Boolean that indicates whether the class is initialized as a celestial point or round body.
- **do_rotate** is a Boolean indicating if the toolkit should rotate the star. This only applies to celestial round stars, which should only comprise of the Sun. Unlike planets or satellites, this rotation is barely noticeable.
- **dist_converter** is the distance converter. It transforms the location of the star.
- **size_converter** is the size converter. It transforms the size of the star.
- **light_converter** is the light converter. It adapts the level of light emitted by the star.
- **center** is a string and **distance** is a integer or a floating point number. They are optional parameters because they are only required for celestial point stars.

The toolkit allows to modify the properties of **observer**, **center** and **distance**, at any point in the code. All the converters used can also be modified.

The instances of the Star class that are initialized as round bodies have the following class functions:

- **build**(***epoch,corrected=False***) creates an ellipsoid in Blender with the name of the Star. This ellipsoid is defined by three radii according to the loaded kernel data. The geometry of the ellipsoid is determined by a predefined number of rings and segments, which can later be increased. The object is moved and rotated to the specified epoch according to loaded kernel data, with the possibility of adding aberration corrections.
- **set_material**(***resolution***) adds the material to the object. First, it checks whether a material with the same name as the Star and from the Blender file resolution.*blend* is already present in Blender and eliminates it. It tries to import from said file the corresponding material and add it to the object, but if the material that the toolkit is looking for is not present in the resolution.*blend* file, it imports and adds to the object the default star material. Finally, the toolkit smooths the faces of the ellipsoid and it modifies the intensity of the emitted light according to the size converter used. It has to be noted that most of the times the user is required to adapt the intensity of the emitted light to suit their needs.
- **to_epoch**(***epoch,keyframe=False,corrected=False***) moves and rotates the star to the designated epoch according to loaded kernel data. It also computes aberration corrections

and adds keyframes for the object if specified.

Instead of obtaining the required data from the loaded kernels using SPICE, the instances of the Star class initialized as point bodies obtain the required data from the data set *StarDict.json*. This data set is available in the toolkit as a dictionary named *STARDICT*, and a list of stars formed with the keys of this dictionary is available as *STARLIST*. The data set contains many types of data of each star, although most of them are redundant as they are not used in the toolkit. Relevant to this work, the relevant data from each star is their name, designation, magnitude, right ascension and declination. *StarDict.json* is created using the code *CreateStarDict.py*, which in turn obtains the data from the file *IAU-CSN.txt* [55]. This file, created by the International Astronomical Union, contains various types of data of the brightest 449 stars in the sky [56]. The frame of reference used is the ICRF, which is internally used in the toolkit as J2000. The location of the stars is established at the epoch J2000, and although the stars are not static in the sky ceiling, their relative movement is very insignificant and can almost be negligible.

Celestial point stars have the same set of functions as celestial round stars but their functioning widely differs:

- **build**(***epoch,corrected=False***) creates the star in Blender as a low-polygon spheroid. Each star is created in a conceptual sphere centered in the location indicated by the property **center** and that has a radius indicated by the property **distance**, which is transformed according to the distance converter. Each star position in this conceptual sphere is determined by the right ascension (RA) and declination (DEC). This is done by transforming the location of the star from spherical to Cartesian coordinates, as shown in equation 15:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \text{distance} \begin{bmatrix} \cos DEC \times \cos RA \\ \cos DEC \times \sin RA \\ \sin DEC \end{bmatrix}. \tag{15}$$

  The size of the star is established as a function of the distance and apparent magnitude, as shown in equation 16:

$$\text{size} = \text{distance}(10^{-3-0.2\times\text{magnitude}}). \tag{16}$$

  This goal of this function is to maintain the relative sizes of stars regardless of the distance used and to relate the size of stars with its apparent magnitude. The reason behind the form of this equation is that the perceived flux density of a star is correlated with the star magnitude as shown in equation 17:

$$\text{flux density} = 10^{-0.4\times\text{magnitude}}. \tag{17}$$

  Some adjustments to the formula have been made in order to obtain a better result in Blender. Instead of multiplying the magnitude by -0.4, the size formula multiplies by -0.2. This is done to make the size of the stars much less dependent on magnitude, so there is a smaller difference between the brightest star, Sirius, which has a magnitude of -1.45, and the faintest star in the catalogue, Atakoraka, which has a magnitude of 12.53. To obtain reasonable sizes for the stars, 3 units are subtracted from the exponent.

- **set_material**(***resolution***) works very similarly to the round star function. First, it checks if a material with the name of the star is present and deletes it. Then, it tries to import a material from the resolution.*blend* file that matches the name of the star. If it is unable to locate it, it uses the default star material. It does not modify the light emitted by the material because point stars are not supposed to illuminate the scene and because the apparent brightness of each star is shown by modifying their size, as modifying the light emitted of these stars in Blender is not useful.

- **to_epoch**(*epoch,keyframe=False,corrected=False*) simply moves the star to its specific location, following the same structure of distance, spherical coordinates, and center. This class function should only be called if the center and the observer are located in two different points or a property of the class has changed. Calling this function in each frame is most of the times not necessary and adds a lot of computing time because of the great number of stars present. If specified, keyframes are added and aberration corrections are applied to the location of the center.

6.9.2 Planet and Moon classes

The Planet class uses the data set *PlanetDict.json*, which is created using the code *CreatePlanetDict.py*. This data set contains, for each planet, the SPICE ids of all the moons that are contained in the SPK kernels, and therefore they are recreated in SPICE. If at some point more kernels with satellite data are used, this data set should be updated with the respective moons. This data set is available in the toolkit as a dictionary named *PLANETDICT*, and a list of planets formed with the keys of this dictionary is available as *PLANETLIST*. As a side note, SPICE considers that Pluto is a planet for backwards compatibility reasons.

The Moon class is not designed to be used in the main code. Instead, the moon class is intended to be used internally by the toolkit. When creating an instance of the Planet class, the toolkit automatically creates an instance of the Moon class for each of its satellites. They share all of the properties, and when a class function is used on a planet, its associated moons also execute their respective class function with the same parameters. The reason behind this approach is that it makes little sense to have satellites without the planet they are orbiting, and therefore it is reasonable that these two types of objects are closely linked.

The instances of the Planet class are initialized with the following attributes:

- **name** is a string identifying the planet. Since it is used to retrieve data in SPICE it can not be an arbitrary name. It must conform to the set of identifiers recognized by SPICE.
- **has_moons** is a Boolean that indicates whether the satellites of this planet should be recreated. There is no problem if this attribute is true but there are no moons on that planet, such as in Mercury or Venus.
- **observer** is a string specifying which is the observer point or body. The toolkit considers that the observer corresponds with the point of reference or origin in Blender. This is shared by its moons.
- **do_rotate** is a Boolean indicating whether the toolkit should rotate the planet and its associated moons.
- **has_rings** is a Boolean indicating whether the rings of this planet have to be recreated.

- **dist_converter** is the distance converter. It transforms the location of the planet and its moons.
- **size_converter** is the size converter. It transforms the size of the planet and its moons.
- **iscycles** is a Boolean indicating whether the Blender scene uses Cycles or Eevee. The choice in the render engine affects how the materials of the planet and moons are treated.

At any point in the main code, the toolkit allows to modify the converters, the properties of **observer** and **iscycles**, and also to obtain a list with all the moons associated with the planet.

The class functions of the instances of the Planet class are the following:

- **build**(***epoch,corrected=False***) creates the planet and its associated moons in Blender as ellipsoids using the data of the three radii retrieved from SPICE. The planets are created with a high polygon count, while the polygon count for moons is lower, although this can be modified. If indicated, the toolkit imports the model of the planet rings from the internally specified file and links it to the planet. If the toolkit cannot locate data about the rotation of a body, the property **do_rotate** is set as false to avoid errors. Finally, the toolkit moves and rotate the planet and its moons the designated epoch according to SPICE data, with aberration corrections if specified.
- **set_material**(***resolution***) adds the material to the planet and its moons. For each body, the toolkit checks if a material with their name and from the file resolution*.blend* is present in Blender, and deletes it. Then, it imports and adds to the object the corresponding material from the specified file. If this material is not present in the file, the toolkit uses a default material. If the planet has rings, it also imports and adds the material of the rings. Finally, the toolkit smooths the faces of the objects and it changes the material node *'Is Cycles?'* of all materials to one if the property **iscycles** is true, and it leaves it at zero if otherwise.
- **to_epoch**(***epoch,keyframe=False,updatelight=True,corrected=False***) moves and rotates the planet and its moons to the corresponding epoch according to SPICE data. If specified, the toolkit adds aberration corrections, adds the relevant keyframes for each body and updates the material node *'Sun direction'* according to the position of the Sun relative to the specific body.

6.9.3 Probe class

The Probe class uses the data set *CamDict.json*, which is created using the code *CreateCamDict.py*. This data set contains the data of the cameras of each spacecraft, such as their name, their location relative to the center of the probe or the focal length. This data set is available in the toolkit as the dictionary *CAMDICT*, and a list of spacecrafts of contained in this dictionary is available as *SPACECRAFTLIST*.

The instances of the Probe class are initialized with the following attributes:

- **name** is a string identifying the probe or spacecraft. It can not be an arbitrary name because it is used to retrieve data from SPICE, and as such, it must conform to the set of identifiers recognized by SPICE.
- **observer** is a string specifying which is the observer point or body. The toolkit considers

that the observer corresponds with the point of reference or origin in Blender.

- **do_rotate** is a Boolean indicating whether the toolkit should rotate the probe.
- **dist_converter** is the distance converter.
- **size_converter** is the size converter.
- **iscycles** is a Boolean indicating whether the Blender scene uses Cycles or Eevee.

At any point in the main code, the toolkit allows to modify the converters, the properties of **observer** and **iscycles**.

The only two class functions of the instances of Probe class are the following:

- **build**(***epoch,tolerance,file='SPACECRAFT',clip_end=1000,clip_start=0.1,corrected=False***) creates the spacecraft in Blender. Instead of creating the mesh in the scene, the toolkit imports the model from the file *SPACECRAFT.blend* and it resizes it with the size converter. There is no need to add any material to the model because Blender automatically does this when importing the model. Then, the probe cameras are added to the spacecraft with the specified clipping range and with the properties established in the data set, which is explained in more detail in subsection 6.10. The toolkit moves and rotates the spacecraft to the corresponding epoch, with aberration corrections if specified. The movement and rotation of the probe takes into account the tolerance, as explained in subsection 4.4. If **iscycles** is false, the toolkit creates a small point light to illuminate the spacecraft. This light is located at 100 metres away from the spacecraft, in the direction of the Sun position. Its power and radius should be adjusted by the user to obtain a good result.

- **to_epoch**(***epoch,keyframe=False,tolerance=0,corrected=False***) moves and rotates the spacecraft to the corresponding epoch, taking into account the specified tolerance, and rotates the probe cameras according to the data of that instant, which is explained in more detail in subsection 6.10. If **iscycles** is false, the toolkit updates the position of the light that illuminates the spacecraft. Finally, the keyframes and aberration corrections are added if specified.

6.10 Probe cameras

The treatment of probe cameras by the toolkit is not simple matter. In a first version, the toolkit obtained from the data set the position and rotation relative to the spacecraft and the focal length of each probe camera. With this data, the toolkit created the different cameras and linked them to spacecraft. Therefore, the cameras moved and rotated jointly with the spacecraft. This had a close to optimal result with Cassini, because the cameras and instruments of this probe were attached to the structure and didn't rotate relative to the spacecraft. However, this method made it impossible to recreate the cameras of the Voyager missions, the reason being that they were mounted on a rotating platform. As such, a new method of treating probe cameras is developed.

In this new method, the toolkit first obtains from the data set the name, identifier, relative location, pointing axis, up axis and focal length of each probe camera. The identifier is mostly used to retrieve data from SPICE. An empty object, that is, an object that Blender does not render, is created with the name *Empty position-identifier* and with its relative position to the probe in accordance with the data set. This empty object is linked to the spacecraft, so that it moves and rotates jointly, as seen in the figure 16, which shows the movement of Empty Position of the Narrow Angle Camera of the Voyager 1 mission. This empty object establishes the position of the camera.

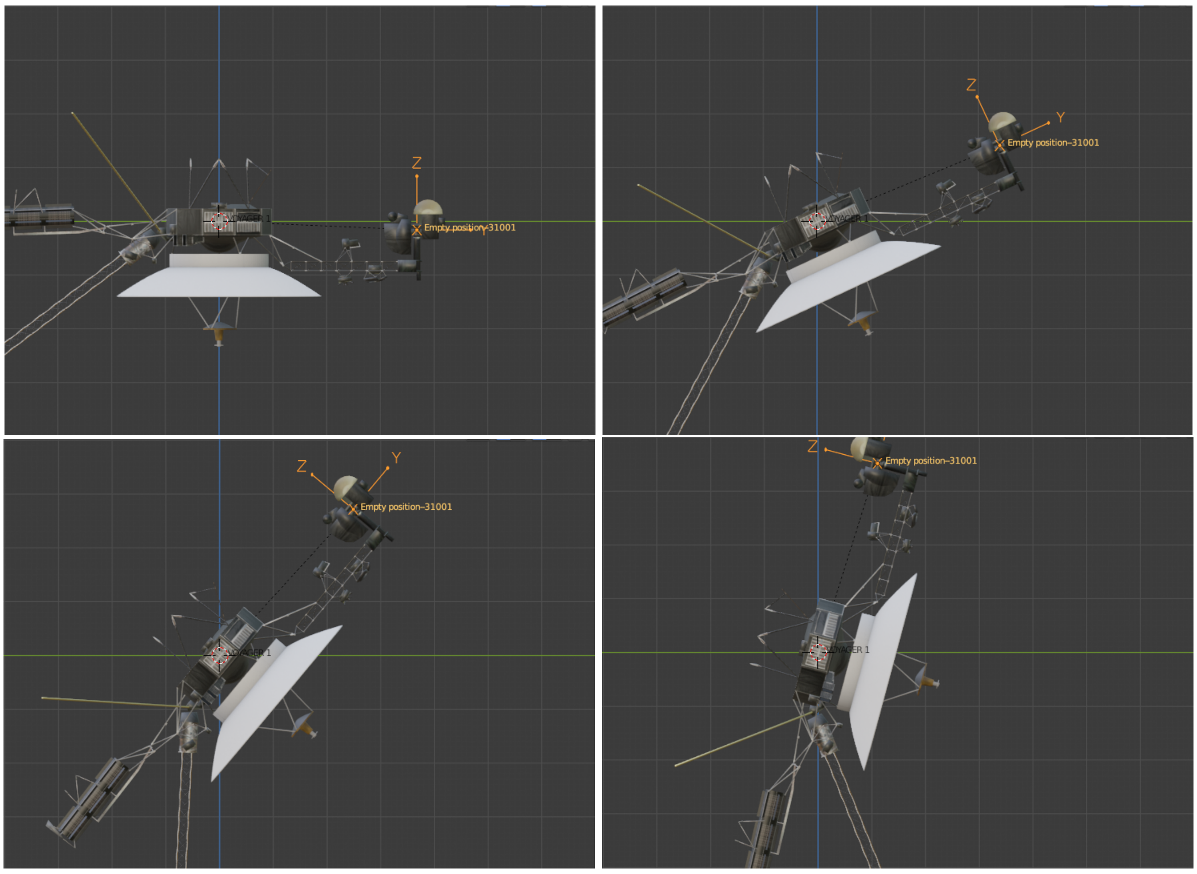


Figure 16: Visualization of probe camera empty position.

Afterwards, a second empty object is created named *Empty rotation-identifier*. This empty object has the object constraint *Copy Location*, the target of which is the Empty Position. This means that the two empty objects share the same location. The main difference between the two lies in that, as seen in the figure 17, the attitude of the Empty Rotation is not affected by the movement of the spacecraft. This empty object establishes the attitude of the camera. For this reason, when the pointing of the camera is updated with the data from SPICE of the appropriate instrument frame, the toolkit rotates the Empty Rotation accordingly. This means that the class function **to_epoch** of the probes does not directly rotate the camera, but instead it rotates this empty object.

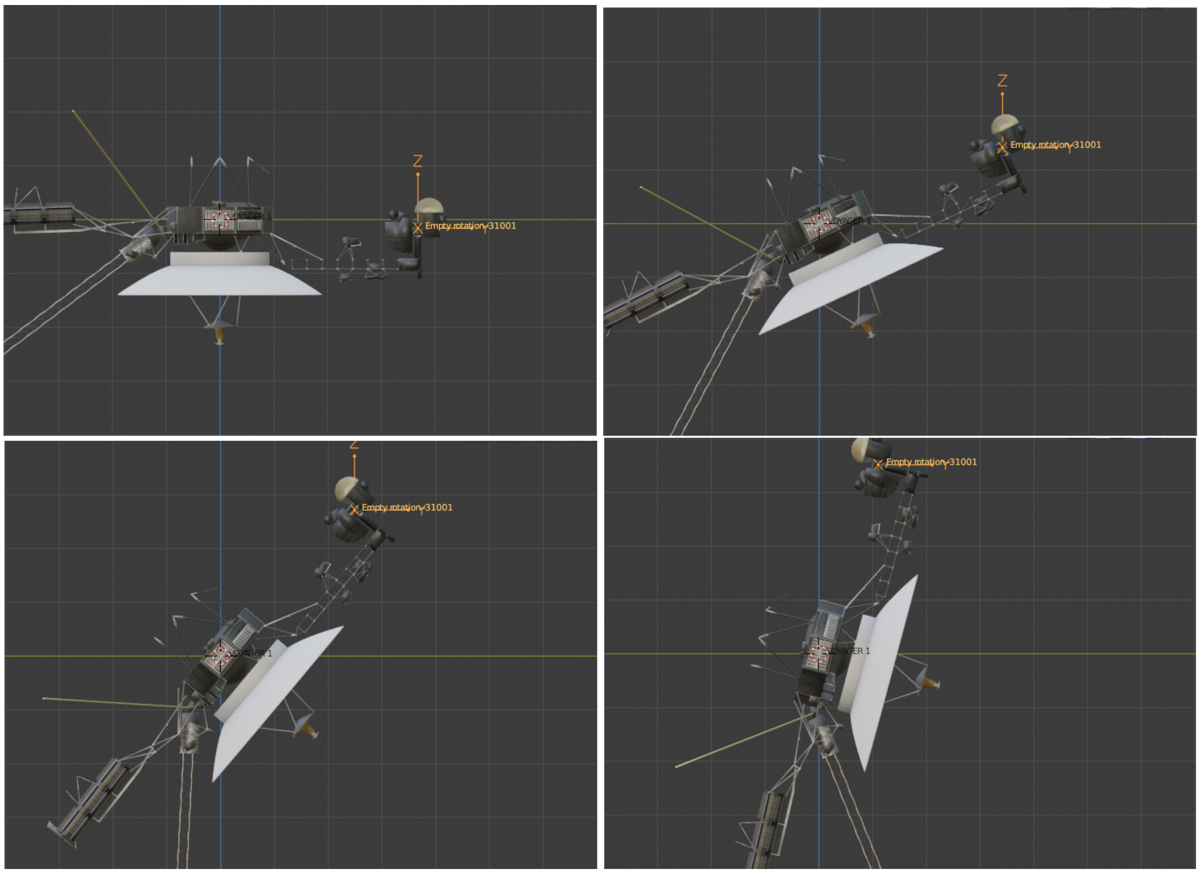


Figure 17: Visualization of probe camera empty rotation.

Finally, the camera is created with the name, position and focal length specified in the data set. Then, it is rotated according to the pointing axis and up axis of the data set. The figure 18 shows the different combinations of the camera axis. Starting from the top left, and following a clockwise pattern, the figure shows the attitude of the camera with the following pointing axis and up axis: +Z and -Y, +Z and +Y, +Y and -Z, +Y and +Z. The axis of the cameras are treated this way because in Blender the cameras always point towards their local -Z axis, and their up axis is always +Y. This is not necessarily true for the cases recreated by the toolkit, and as such it is specified for every instrument. In the end, the camera is linked to the Empty Rotation. Therefore, the camera is located at the position of the Empty Position, and its attitude is that of the Empty rotation.

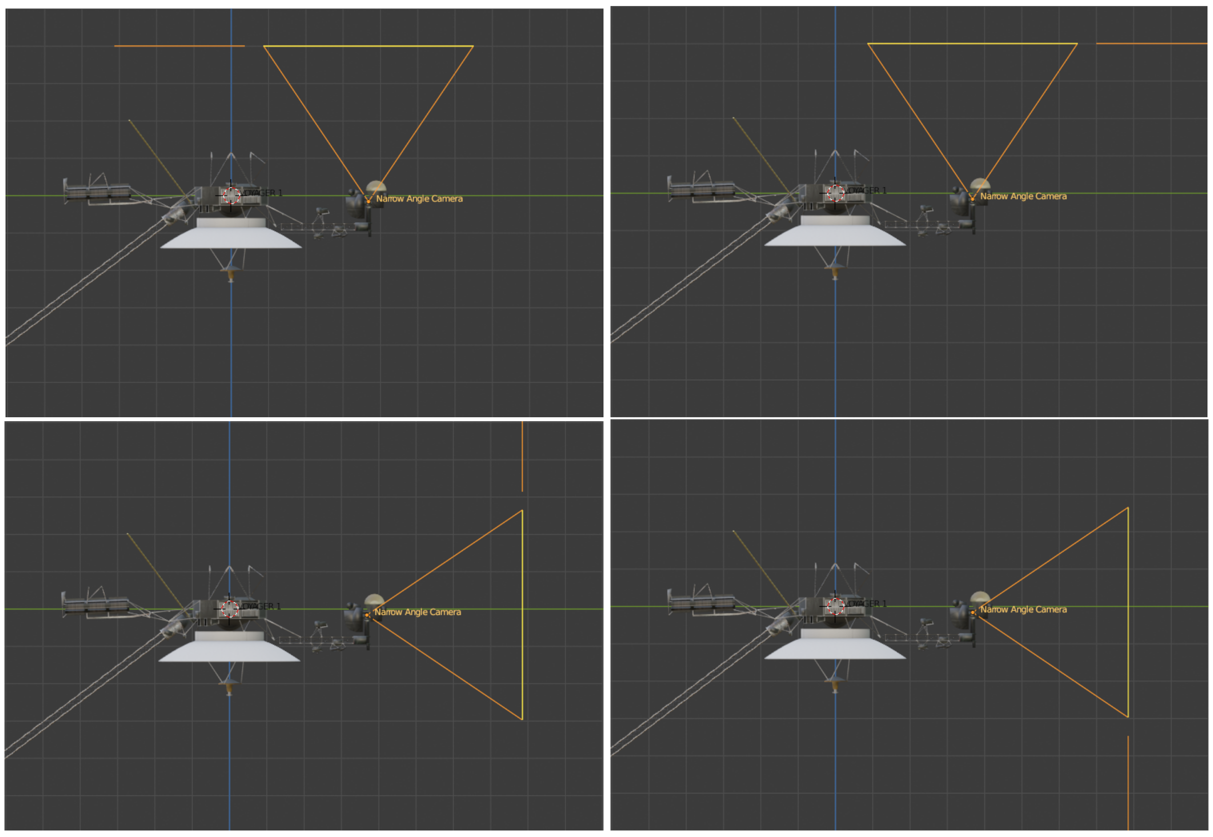


Figure 18: Visualization of probe camera different axis.

This method of dealing with the probe cameras, although it is more complicated, it offers two substantial improvements. First of all, it offers the possibility of recreating both spacecrafts with fixed, such as the Cassini, and with movable instruments, such as the Voyager missions. And secondly, it's level of precision is very high. This is because the position of the camera relative to the spacecraft is very close to that of the real instrument, and to compute the direction the camera is aiming the toolkit obtains information in SPICE of its instrument frame, instead of just rotating the spacecraft.

7 Example cases

In this section, a set of cases are presented in order to better explain how does the toolkit work and to show examples of what it can achieve. The rendering of the results with high resolution materials have been possible thanks to Professor Miró.

7.1 Earth rising

The goal of this first case is to recreate the iconic image "Earth Rising over the Moon's Horizon", seen on the left of figure 19, next to the recreated picture. This image was taken from the Apollo 11 spacecraft on July 20, 1969. The exact instant is determined by doing some iterations with the code so that the position of the Earth in Blender corresponded with that of the picture. There aren't any Apollo 11 kernels provided by NAIF, therefore, for this case in particular, no spacecraft trajectory is used.

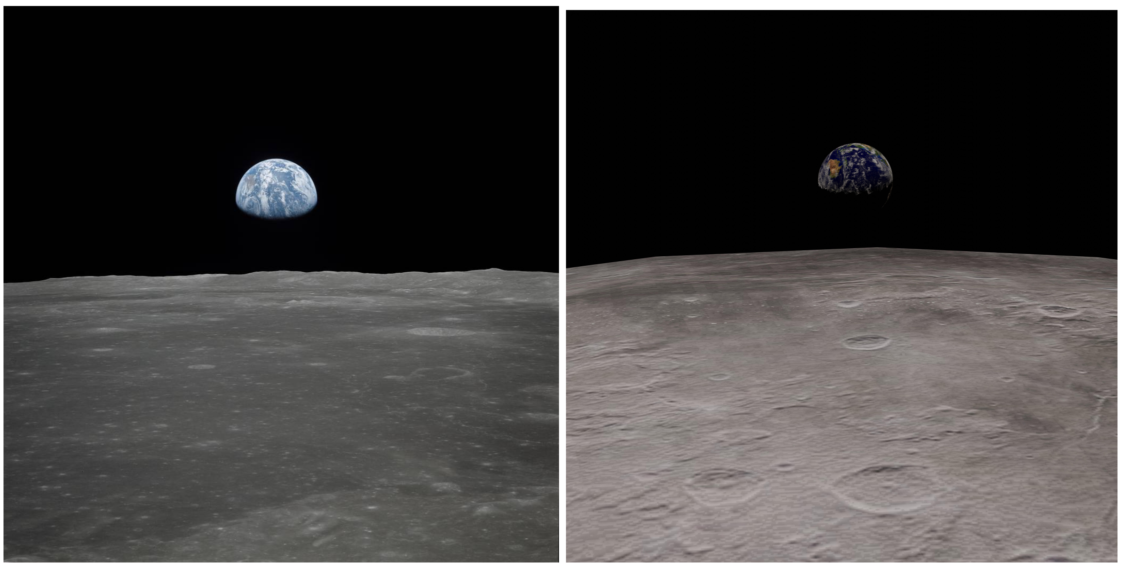


Figure 19: Earth Rising over the Moon's Horizon[57] (left), and recreated image (right).

The main code used for this case is contained in the *EarthRising.py* file. The set of properties that define the characteristics of the result and that are used throughout the code are shown in the table 3. The clipping range is large enough to observe the different objects, the Moon set as the observer and high resolution materials are used. Note that no aberration corrections are applied in this case.

is_cycles	True
clip_end	10^7
clip_start	10^{-2}
show_object_name	True
observer	MOON
correctedState	False
materials	HIGH_RES

Table 3: Earth Rising properties.

The functioning of *EarthRising.py* is represented with pseudo code in algorithm 3. First of all, the code imports the modules used, which consist of *Blenthon* and the Blender API *bpy*. Then, the properties described above are defined in the code. The toolkit loads the kernels into memory, cleans the scene in Blender and changes its background.

Since the goal of this case is to recreate only one image, the number of frames is set to 1 and the single epoch corresponding with 1969/07/20 23:00:00.00 is obtained. This epoch, denoted by *instant* in the algorithm, is in UTC and corresponds with the approximate time the real image was taken.

The Sun is created as a celestial round Star, using the *earthradius* conversion factor for both the size and distance, which corresponds with $6,371$km. Then, it is built in Blender, the material is added, and it is moved to the corresponding epoch.

Afterwards, the Earth is created as a Planet, using the same converters as the Sun. As such, the resulting scene is in real scale. When creating the Earth, *has_moons* is set as true, so that the Moon is created automatically. Then, the Earth alongside the Moon are built in Blender, the materials are added, and they are moved to the corresponding epoch.

The objects that represent the Earth and the Moon are subdivided so that they have a better quality in the render. The camera is created using the coordinates (loc) and attitude (rot) determined by manually angling the camera, alongside the appropriate camera lens. Finally, the code finishes the toolkit to get the resulting final scene.

Algorithm 3: Representation of *EarthRising.py* with pseudo code.

```
Result: The image Earth Rising is recreated
import modules
define properties
init_spice(), clean(), set_background()
set_frames(1)
$epoch$ = timeperiod($instant$)
initialize $Sun$ as celestialRound Star
$Sun$.build, $Sun$.set_material, $Sun$.to_epoch($epoch$)
initialize $Earth$ as Planet
$Earth$.build, $Earth$.set_material, $Earth$.to_epoch($epoch$)
subdivide_object($Earth$,$Moon$)
add_camera($loc$,$rot$)
finsih()
```

After executing the code, some aspects of the Blender file had to be modified to obtain the desired result, such as adjusting the light emission of the sun to provide a good level of lighting. Before rendering, the render and output properties require some user input, such as the number of samples, denoising, resolution or compression, just to name a few. Setting the correct render parameters can be rather cumbersome and is outside the scope of this project. As a final note for this case, the figure 20 shows a comparison of the same scene rendered with Cycles (right) and with Eevee (left).

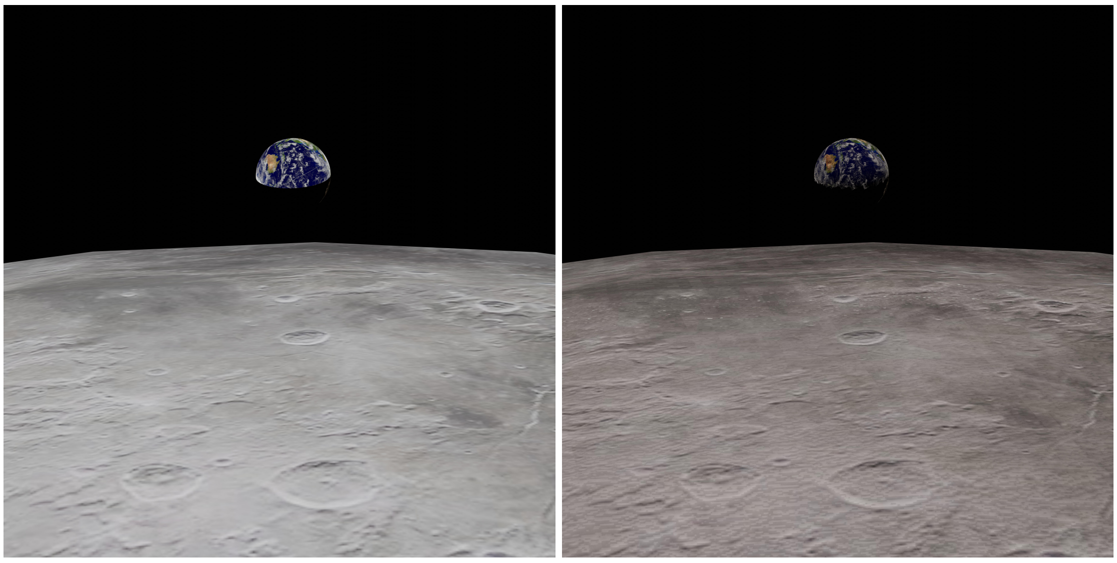


Figure 20: Comparison of the recreated image Earth Rising with Eevee (left) and Cycles (right).

7.2 Inner Solar System

The goal of this example is to create an animation visualizing the inner solar system planets as they move through a whole year. The main code used is named *InnerSolarSystem.py*. The properties of this scene are shown in the table 4. These are very similar to the previous case, the differences being that the observer body is the Sun and that stars are present. The stars are located at a distance of 10^6 units, and their center corresponds with the observer, the Sun.

is_cycles	True
clip_end	10^7
clip_start	10^{-2}
show_object_name	True
observer	SUN
correctedState	False
materials	HIGH_RES
stars_center	SUN
stars_distance	10^6

Table 4: Inner Solar System properties.

The functioning of *InnerSolarSystem.py* is represented in algorithm 4. Similarly to the previous case, the code imports the modules *Blenthon* and *bpy* and the properties are defined. Then, the kernels are loaded, the scene is cleared, and the background is set. The number of frames is set to twelve frames per day, in other words each day lasts half a second at 24fps. The total number of frames corresponds with $365 \times 12 = 4380$. Then, the set of epochs equally distributed is obtained from the *starttime*, which is arbitrary set to 2019/01/01 00:00:00.00, to the *endtime*, which is set to one year later, at 2020/01/01 00:00:00.00. This way, a correspondence is established between each frame and an epoch.

The Sun is created as celestial round Star, using the Earth to Sun distance converter, which corresponds with a factor of 1.496×10^8, and the Earth to Moon size converter, which corresponds with a factor of 3.844×10^5. The size converter factor is multiplied by 10, so in the end the Sun size factor is 3.844×10^6. This means that the scale used in the size is about 40 times bigger than the distance scale. The Sun is built in Blender and its material is added.

The inner planets, that is, Mercury, Venus, the Earth and Mars, are created as planets and added to a list for easier management. Similarly to the Sun, they use the Earth to Sun as the distance converter factor, and the Earth to Moon as the size converter factor. The size converter factor is divided by 5, so in the end the planet size factor is 7.688×10^4. This means that the size scale of the planets is 2000 times bigger than the distance scale. Relative to the Sun, the size of the planets is 50 times bigger. The attribute *has_moons* is set as false, so none of the moons of Mars or Earth are recreated. The planets are built in Blender and their material added.

Each sky star in the list of stars is created as a celestial point star. They use the distance and size converter "real", which has a factor of 1. This means that the stars are located at a distance of 10^6 blender units, well below the clip end value but still far away enough that they appear to be points in the sky. The stars are built in Blender and their material added. Since the observer and the stars center coincide through the whole animation, there is no need to move the stars at any frame.

The animation is performed by cycling through every epoch. At each epoch, Blender is taken to its associated frame. Then, all the planets are moved to their position and attitude at that epoch. There is no need to move the Sun because it is the observer body and its rotation can not be appreciated. Finally, a camera is added with location *loc* and attitude *rot*. This camera shows the whole animation of the scene from a fixed point of view. The toolkit is ended, and some adjustments to the result are performed to obtain a better result.

Algorithm 4: Representation of *InnerSolarSystem.py* with pseudo code.

```
Result: The animation of the Inner Solar System is recreated
import modules
define properties
init_spice(), clean(), set_background()
set_frames($365 \times 12$)
epochs = timeperiod(starttime,endtime)
initialize Sun as celestialRound Star
Sun.size_factor*=10
Sun.build, Sun.set_material
for planet in PLANETLIST[:4] do
    initialize planet as Planet, add planet to planets
    planet.size_factor/=5
    planet.build, planet.set_material
for star in STARLIST do
    initialize star as Celestial Point Star
    star.build, star.set_material
for frame, epoch in enumerate(epochs) do
    to_frame(frame)
    for planet in planets do
        planet.to_epoch(epoch)
add_camera(loc,rot)
finsih()
```

The figure 21 shows four frames of the resulting video, which correspond with the following dates: top left 2019/01/01, top right 2019/05/02, bottom left 2019/09/01, bottom right 2020/01/01. The whole render of the animation is available online [58].

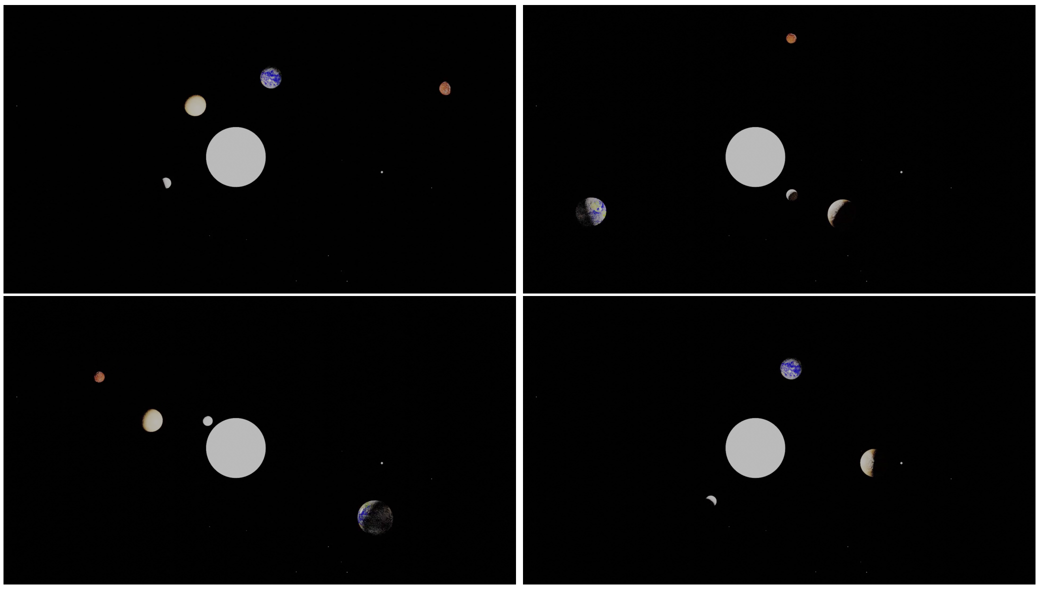


Figure 21: Frames of the recreated animation of Inner Solar System, which correspond with: top left 2019/01/01, top right 2019/05/02, bottom left 2019/09/01, bottom right 2020/01/01.

7.3 Constellation

The goal of this case is to recreate part of the night sky and compare the recreated positions of the stars with their actual location. The figure 22 shows a section of the sky at 2020/05/26 21:55:00.00 as seen from Barcelona. On top, the image obtained using the web version of Stellarium, a free open source planetarium [59]. On the bottom, the recreated image.

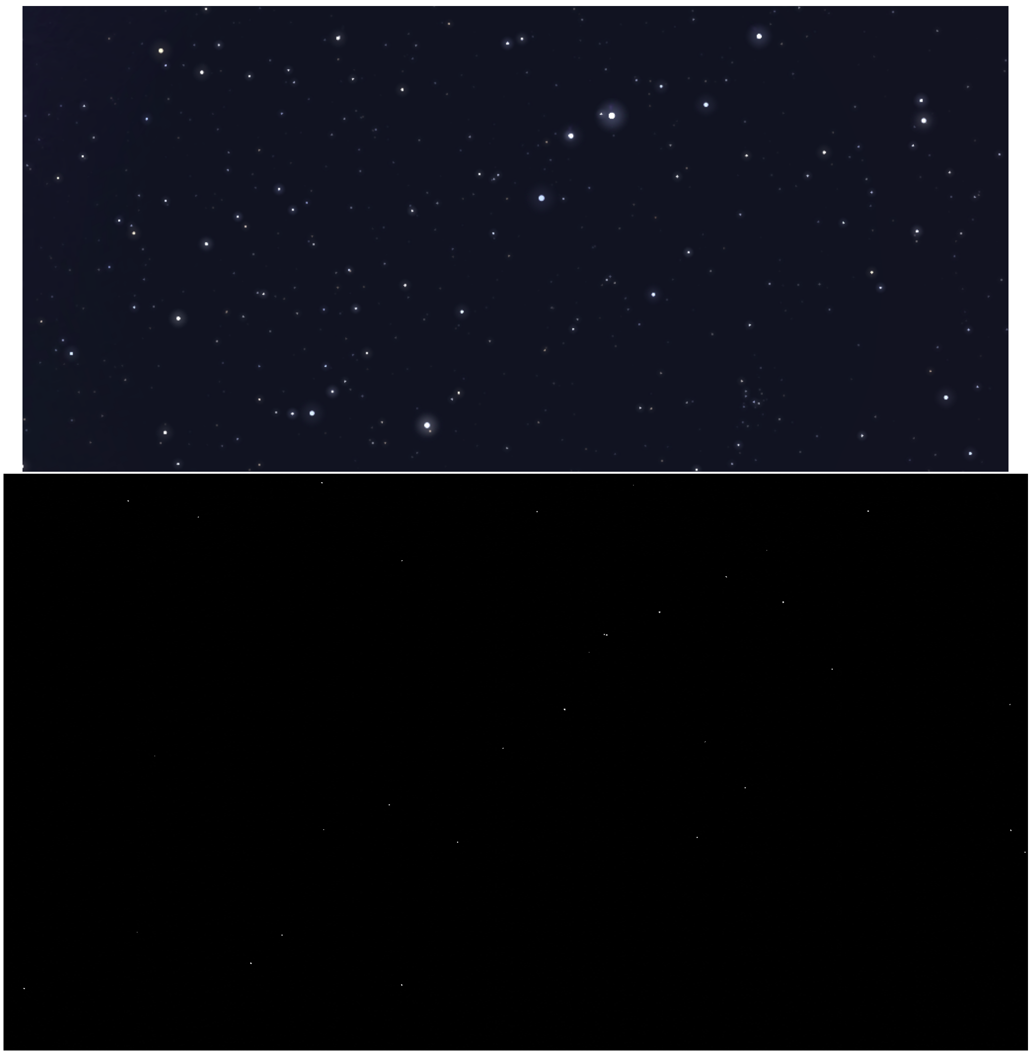


Figure 22: Comparison of recreated Constellation with Stellarium (top) and with *Constellation.py* (bottom).

The main code used is named *Constellation.py*, and the properties of the scene are shown in table 5. Compared to the previous case, the main difference is that Eevee render is used alongside low resolution textures because this case just renders the stars. The observer and stars center is set as the Earth, because this is the point of view of the night sky.

The functioning of *Constellation.py* is represented in algorithm 5. Similarly to the previous cases, the code imports the modules *Blenthon* and *bpy*, and the properties are defined. The kernels are loaded, the scene cleaned, and the background set. Only one frame is needed, which corresponds with the epoch 2020/05/26 21:55:00.00, denoted as *instant*.

The Sun is created as a celestial round Star, using the Earth radius as the size and distance converters. The Sun is built, its material added, and it is moved to the corresponding epoch. The Earth is created as a Planet, with the attribute *has_moons* set as true so the Moon is recreated. It uses the same converters as the Sun. The Earth is built in Blender, its material added, and it is moved to the corresponding epoch. All the stars of the sky present in the catalogue are created

is_cycles	False
clip_end	10^7
clip_start	10^{-2}
show_object_name	True
observer	EARTH
correctedState	False
materials	LOW_RES
stars_center	EARTH
stars_distance	10^6

Table 5: Constellation properties.

as celestial point stars. They are built in Blender and their material set.

The camera used is added with location *loc* and attitude *rot*. It is located approximately on Barcelona, pointing towards a specific section of the sky. Finally, the toolkit is ended, and the resulting image is obtained.

Algorithm 5: Representation of *Constellation.py* with pseudo code.

```
Result: The section of the night sky is recreated
import modules
define properties
init_spice(), clean(), set_background()
set_frames(1)
epoch = timeperiod(instant)
initialize Sun as celestialRound Star
Sun.build, Sun.set_material, Sun.to_epoch(epoch)
initialize Earth as Planet
Earth.build, Earth.set_material, Earth.to_epoch(epoch)
for star in STARLIST do
    initialize star as Celestial Point Star
    star.build, star.set_material
add_camera(loc,rot)
finsih()
```

The figure 23 shows a comparison of the recreated image and the same section of the sky using Stellarium. Red lines have been added between the same set of stars, in order to help visualize the similarities between the two pictures.

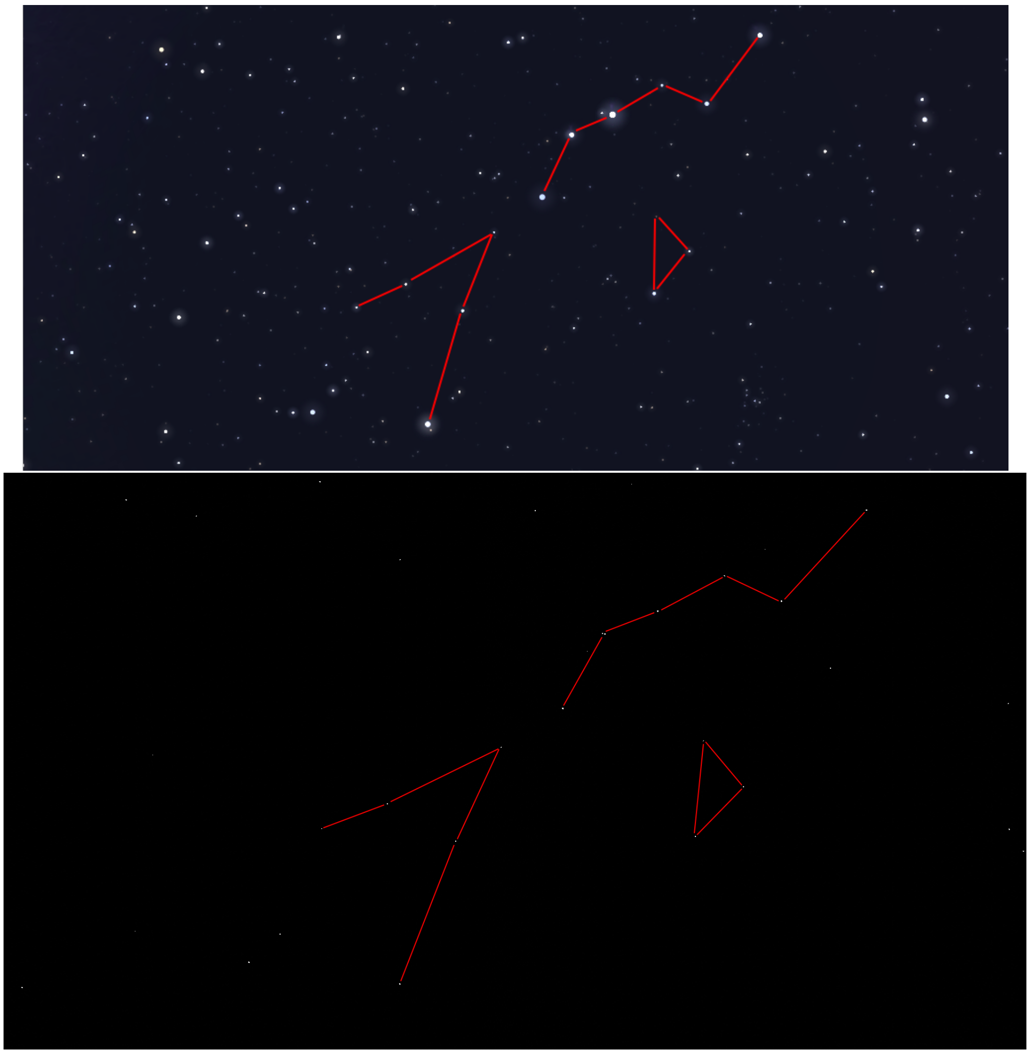


Figure 23: Comparison of constellations from Stellarium (top) and recreated with *Constellation.py* (bottom)

7.4 Bright Moons

The goal of this section is to recreate the image Bright Moons [60]. This image, seen on figure 24 on the left, was taken by Cassini on April 25, 2011, shows, from left to right, Dione, Rhea and Enceladus. The recreated image is shown on the right of figure 24.

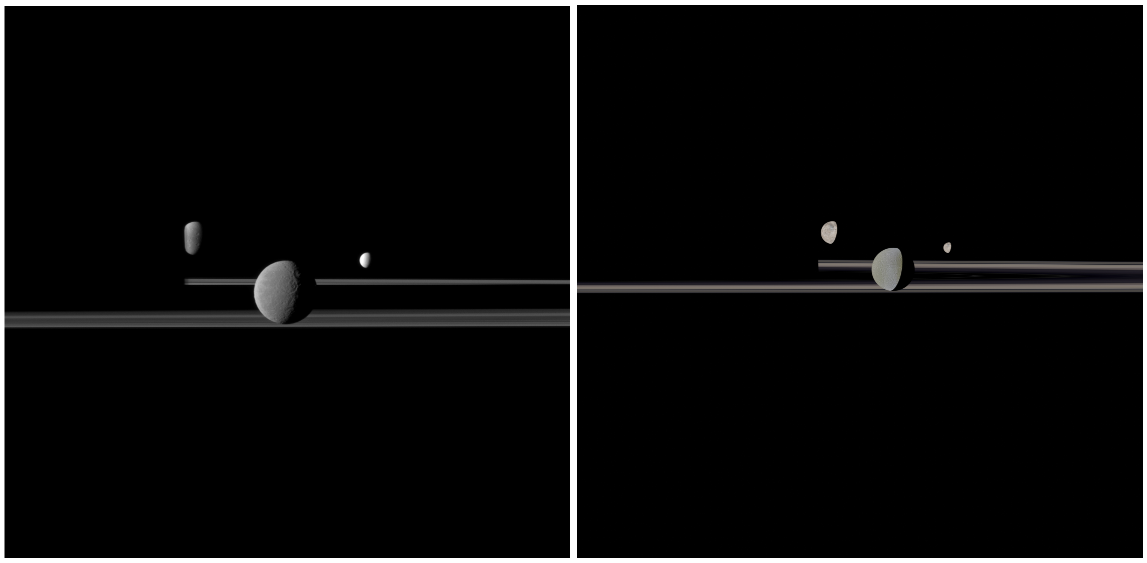


Figure 24: Comparison of real Bright Moons image (left) and recreated (right)

The main code used to obtain this image is named *brightMoons.py*. The properties of the recreated scene are shown in table 6. The recreated spacecraft is Cassini, which also serves as the observer. Aberration corrections are applied to obtain an image closer to the real one. The render is done with Eevee and high resolution materials.

is_cycles	False
clip_end	10^7
clip_start	10^{-2}
show_object_name	True
spacecraft	CASSINI
observer	CASSINI
planet	SATURN
correctedState	True
materials	HIGH_RES

Table 6: Bright Moons properties.

The functioning of *brightMoons.py* is represented in algorithm 6. First, the modules *Blenthon* and *bpy* are imported, and the properties are defined. The kernels are loaded, the scene cleaned, and the background set.

Using *findAllInstances*, the toolkit looks in the *RESSLISTS* for Cassini images during Saturn encounter that were taken by the Narrow Angle Camera during April 25, 2011, which is defined by *starttime* and *endtime*. The images that fit this criteria, which consist of 20 instances, result in a set of epochs. The length of this list is set as the number of frames.

The Sun is created as a celestial round Star, using the Earth radius as the size and distance converter. It is built in Blender and its material set. Saturn is created as a Planet with *has_moons* and *has_rings* set as true. Using the same converters as the Sun, Saturn is built and its material added.

Cassini is created as probe, using the same distance converter, but using the size converter "real". This means that the probe is $6,371$ times bigger than in real scale. The reason behind this is to lower the clip range needed to obtain the scene. The tolerance is set to 0 because the data of Cassini is located in multiple kernels and is very complete. The spacecraft is built in Blender.

The moons seen on the image, and the rings of Saturn, are subdivided to increase their render quality. At each frame, the Sun, Saturn alongside its moons, and Cassini, are taken to the corresponding epoch. The toolkit is finished, and some modifications to the scene are performed to obtain the final result. The ninth out of a total of twenty frames is the target image.

Algorithm 6: Representation of *BrightMoons.py* with pseudo code.

```
Result: The image Bright Moons is recreated
import modules
define properties
init_spice(), clean(), set_background()
epochs = findAllInstances(from=starttime,to=endtime,instrument=Narrow Angle)
set_frames(length(epochs))
initialize Sun as celestialRound Star
Sun.build, Sun.set_material
initialize Saturn as Planet
Saturn.build, Saturn.set_material
initialize Cassini as Probe, tolerance=0
Cassini.build
subdivide(Dione,Rhea,Enceladus,Rings)
for frame, epoch in enumerate(epochs) do
    to_frame(frame)
    Sun.to_epoch(epoch)
    Saturn.to_epoch(epoch)
    Cassini.to_epoch(epoch)
finsih()
```

As seen on figure 24, the recreated image on the right is really close to the real one. Both the location of the different bodies and the line of day on the moons seem to be very precise. However, it can be appreciated that the leftmost moon, Dione, is not partially hidden by Saturn. This is most likely because the toolkit does not recreate the atmospheres of the planets, and it can be observed that the rings are hidden by what looks like Saturn's atmosphere. However, this has not been proven, and there might be another unknown reason for this inaccuracy.

7.5 Pan flyover

The goal of this case is to recreate the image Pan In the Middle [61]. This image was taken by Cassini with the narrow angle camera on June 25, 2012. It shows Pan, a small irregular moon of about 30km, inside the Encke Gap, a small overture of in the rings of Saturn of only 325km in width. To give an idea of the scale of this values, the rings span about $140,000$km from the center of Saturn. In other words, the moon Pan is about the size of a small island like Ibiza, the Encke Gap is about one fifth the size of the moon, and the total span of the rings is about the distance between the Earth and the moon.

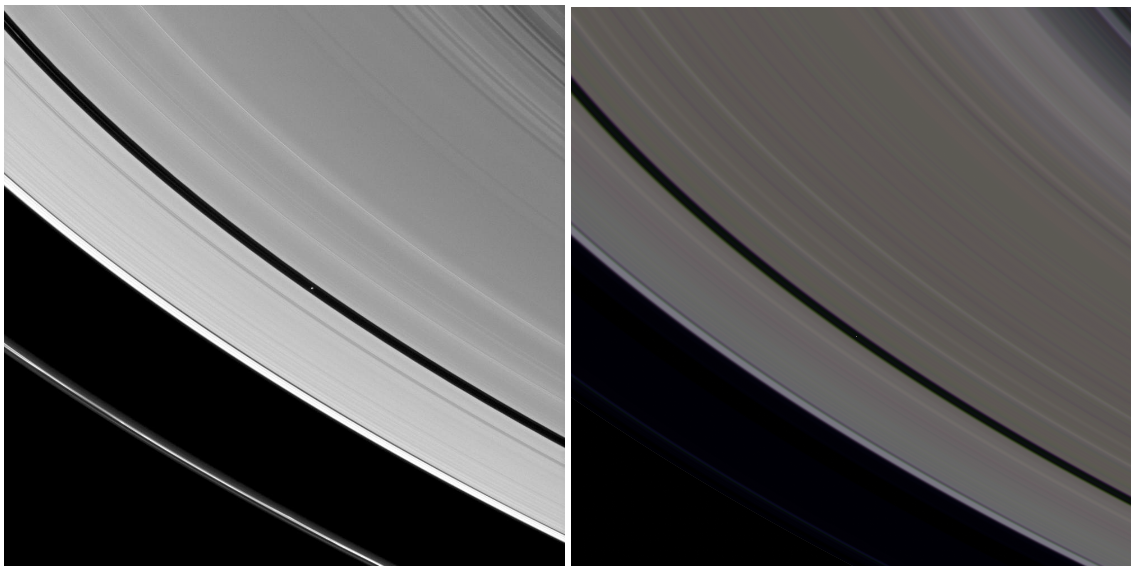


Figure 25: Comparison of real Pan In the Middle image (left) and recreated (right).

The main code used to obtain this image is named *panFlyover.py*. The properties of the recreated scene are shown in table 7. The properties used for this case are the exact same as in the previous case.

is_cycles	False
clip_end	10^7
clip_start	10^{-2}
show_object_name	True
spacecraft	CASSINI
observer	CASSINI
planet	SATURN
correctedState	True
materials	HIGH_RES

Table 7: Pan Flyover properties.

The functioning of *PanFlyover.py* is shown in algorithm 7. The code is really similar to the previous case, in subsection 7.4. As such, instead of repeating the explanation of the code word by word, only the differences between the two codes are presented.

The size and distance converters "real" are used for all objects. This means that for this case, all bodies are in real scale, and that one unit in Blender is equal to one kilometer in real life. Also, only the Saturn rings are subdivided because they are the only body observable in the image, a aside from Pan. However, Pan is so small that it makes little difference to subdivide the moon.

The function *findAllInstances* is used to obtain all images taken by Cassini from 06/01/2012 to 06/30/2012, that had the moon Pan as the target. A total of 10 instances were found to fit this criteria. The tenth frame in the scene corresponds with the target image.

Algorithm 7: Representation of *PanFlyover.py* with pseudo code.

```
Result: The image of Pan In the Middle is recreated
import modules
define properties
init_spice(), clean(), set_background()
epochs = findAllInstances(from=starttime,to=endtime,target=PAN)
set_frames(length(epochs))
initialize Sun as celestialRound Star
Sun.build, Sun.set_material
initialize Saturn as Planet
Saturn.build, Saturn.set_material
initialize Cassini as Probe, tolerance=0
Cassini.build
subdivide(Rings)
for frame, epoch in enumerate(epochs) do
    to_frame(frame)
    Sun.to_epoch(epoch)
    Saturn.to_epoch(epoch)
    Cassini.to_epoch(epoch)
finsih()
```

The recreated image, shown on the right of figure 25, displays a high degree of precision. It can be appreciated the high degree of similarities in the Saturn rings between the real and recreated images. Moreover, although it is somewhat difficult to see, the moon Pan fits perfectly inside the Encke gap. The precision required to obtain this result is very high, as the Blender mesh that models of the rings has to perfectly coincide with the texture of the rings.

7.6 Cassini Insertion

The goal of this case is to recreate the insertion of Cassini into the orbit of Saturn. Instead of obtaining a set of images taken by a mission, this case aims to visualize the maneuvers of the spacecraft and its trajectory. The main code used to obtain this is named *InsertionCassini.py*. The properties of the scene are shown in table 8, and are very similar to the previous case, the only difference being that this case uses low resolution materials.

is_cycles	False
clip_end	10^7
clip_start	10^{-2}
show_object_name	True
spacecraft	CASSINI
observer	CASSINI
correctedState	True
materials	LOW_RES

Table 8: Cassini Insertion properties.

The functioning of *InsertionCassini.py* is shown in algorithm 8. The module *Blenthon* is imported and the properties are defined. The toolkit loads the kernels, clears the scene, and sets the background. The number of frames is set to 240, so that rendering at 24fps results in a video of 10 seconds.

The start and end time of the recreation are established respectively as one hour before and three hours after *instant*, which is an epoch in ET and has to be converted to UTC. *instant* corresponds with the epoch that Cassini started insertion maneuvers. First, Cassini oriented the high gain antenna towards the rings in order to protect itself against impacts, and then it reoriented itself to start the insertion propulsion. The exact epoch and this information was provided by Daniel Regené and Daniel Cantos, and it is consistent with the result obtained.

Similarly to previous cases, the Sun, Saturn and Cassini are created, built, and their material set. They all use the size and distance converters "real", although the converter size of Cassini is divided by 100 in order to reduce the required clipping range. This means that all distances and sizes are in real scale except Cassini, which is 100 times bigger than in real life. Saturn and its rings are subdivided for better quality.

The animation is performed for each frame, and a fixed camera is added with location *loc* and attitude *rot*. Finally, the toolkit is ended, and as usual, some modifications to the scene are performed in order to obtain a better result.

Algorithm 8: Representation of *CassiniInsertion.py* with pseudo code.

```
Result: The animation of Cassini Insertion is recreated
import modules
define properties
init_spice(), clean(), set_background()
set_frames(240)
starttime=epoch_2_utc(instant-3600 × 1)
endtime=epoch_2_utc(instant+3600 × 3)
epochs = timepriod(starttime,endtime)
initialize Sun as celestialRound Star
Sun.build, Sun.set_material
initialize Saturn as Planet
Saturn.build, Saturn.set_material
initialize Cassini as Probe, tolerance=0
Cassini.size_factor/=100
Cassini.build
subdivide(Rings,Saturn)
for frame, epoch in enumerate(epochs) do
    to_frame(frame)
    Sun.to_epoch(epoch)
    Saturn.to_epoch(epoch)
    Cassini.to_epoch(epoch)
add_camera(loc,rot)
finsih()
```

The figure 26 shows four frames of the resulting animation. The two images on the top show the approach of Cassini. The bottom left shows the Cassini facing its high gain antenna towards the rings, and in the bottom right image the spacecraft orientates its engines in the same direction as its trajectory to perform the insertion propulsion. The full video is available online [62].

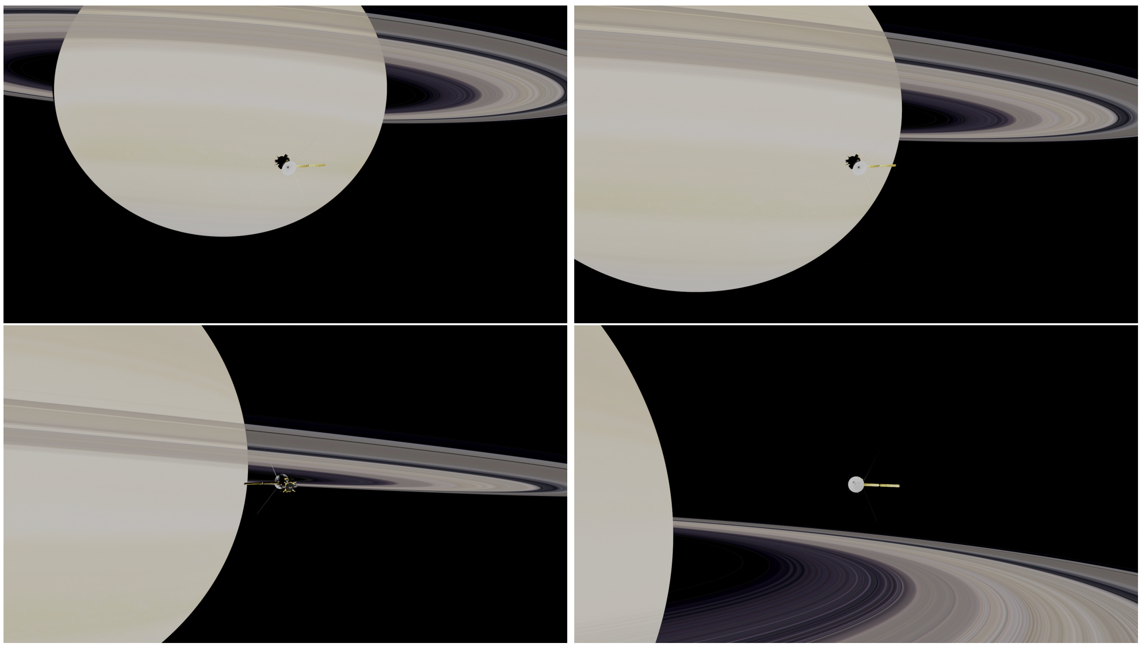


Figure 26: Set of images from the animation of Cassini Insertion.

7.7 Eruption at Io

The goal of this last case is to recreate the image Volcanic Eruptions on Io [63]. This image was taken by Voyager 1 on March 8, 1979. It shows the Jupiter satellite Io, as a volcanic eruption is taking place. It is an iconic image because it was the first time volcanic activity was observed outside the Earth. Figure 27 shows on the left, the original image taken by the mission, and on the right the recreated one.

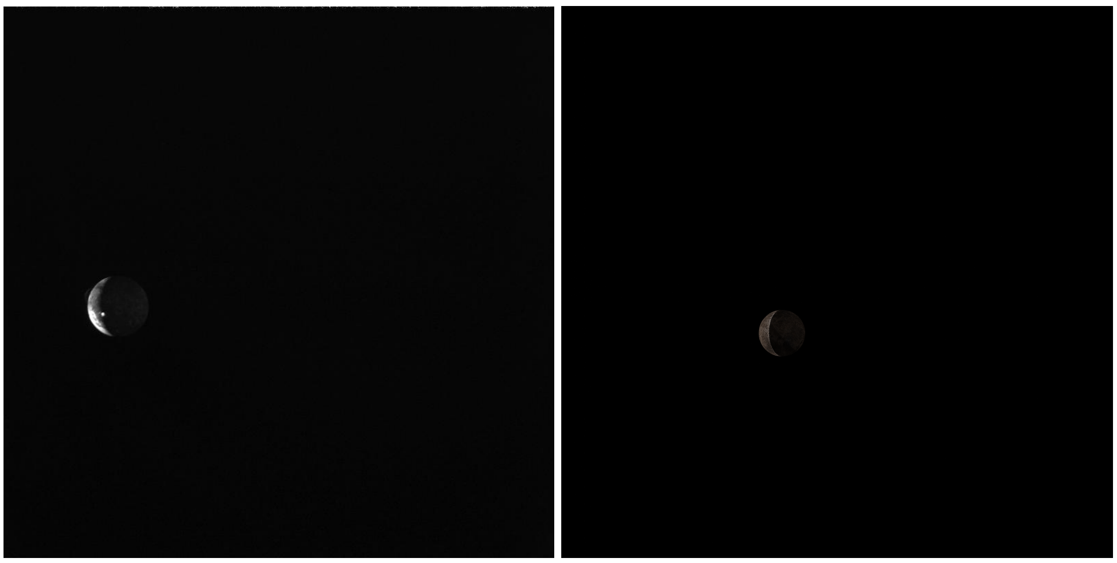


Figure 27: Comparison of Volcanic Eruptions on Io (left) and the recreated image (right).

The main code used to obtain this image is named *eruptionAtIo.py*. The properties of the scene are shown in table 9. The differences relative to the previous case are that this scene uses Cycles with high resolution materials, and that the spacecraft recreated is the Voyager 1, which also serves as the observer.

is_cycles	True
clip_end	10^7
clip_start	10^{-2}
show_object_name	True
spacecraft	VOYAGER 1
observer	VOYAGER 1
correctedState	True
materials	HIGH_RES

Table 9: Eruption At Io properties.

The functioning of *eruptionAtIo.py* is shown in algorithm 9. The *Blenthon* module is imported and the properties defined. The kernels are loaded, the scene cleared and the background set. A set of epochs are obtained that correspond with images taken by Voyager 1 on 1 on March 8, 1979, that target Io. Only one instance fits the criteria.

Similarly to previous cases, the Sun, Jupiter, the stars of the sky and the probe are initialized, created in Blender, and their material set. However, in the end no star was visible in the resulting image. They all use the size and distance converters "real", and the size of the spacecraft is increased by a factor of 100 to limit the clipping range. Only Io is subdivided. All objects are moved to the corresponding epoch and the toolkit is ended. Some modifications to the scene are performed.

Algorithm 9: Representation of *eruptionAtIo.py* with pseudo code.

```
Result: The image of Volcanic Eruptions on Io is recreated
import modules
define properties
init_spice(), clean(), set_background()
epochs = findAllInstances(from=starttime,to=endtime,target=IO)
set_frames(length(epochs))
initialize Sun as celestialRound Star, Sun.build, Sun.set_material
initialize Jupiter as Planet, Jupiter.build, Jupiter.set_material
for star in STARLIST do
    initialize star as Celestial Point Star, star.build, star.set_material
initialize Voyager1 as Probe, tolerance=0
Voyager1.size_factor/=100, Voyager1.build
subdivide(Io)
for frame, epoch in enumerate(epochs) do
    to_frame(frame)
    Sun.to_epoch(epoch), Jupiter.to_epoch(epoch), Voyager1.to_epoch(epoch)
finsih()
```

The recreated image, shown on the right of figure 27, is very close to the real image taken by

Voyager 1. Although the eruption itself is not recreated in Blender, the position of the line of day on Io seems to precisely coincide, and the morphology of Io seems to be similar to the real image too. This case also shows that images taken with cameras on a moving platform, which is the case of Voyager 1, can be recreated the same way as images taken with fixed cameras, such as in the cases with the Cassini.

Conclusions

The aim of this project has been successfully achieved. A toolkit has been developed that can precisely recreate the trajectory of spacecrafts using NASA SPICE and Blender, and as such, help in their visualization. To achieve this, more than 2500 lines of Python code in 21 different files have been written. The toolkit is able to faithfully recreate spacecraft trajectories, provide accurate depictions of the observations by probe instruments and render animations of the solar system bodies, among other possible applications. Moreover, not only is it capable of recreating the Sun and every planet plus Pluto, but the toolkit is able to recreate the missions Cassini, Voyager 1 and 2, up to 66 satellites and 449 of the brightest stars. The toolkit is also capable of modeling planet rings and spacecraft instruments, both fixed and movable. Original methods have been created for this toolkit, such as the treatment of the probe cameras, the Eevee lighting method or the process of material handling. The toolkit has also developed workarounds to Blender limitations, like the unit conversion method, which uses converters that are easily modifiable so the clip range is easily limited, or the Blender lighting method, which instead uses the emission of light instead of reflection. It also offers a high degree of flexibility and the users can easily adapt the code to suit their needs. The final version is open-source and free to use or modify for anyone interested.

The toolkit has been successfully tested in two different Linux computers. There has been no observable difference between the results obtained with the two computers. Moreover, Blender is cross-platform and actively supported on all major operating systems. There is no reason to doubt that the deployment of the developed toolkit would be successful in any computer, as long as it fulfills the minimum system requirements [64].

More than 20 Blender materials of high and low resolution have been create for the stars, the planets and the major satellites, in addition to the material of the rings of Saturn. These materials have a high degree of quality. They use free high precision maps of the solar system bodies obtained from NASA and other sources. The node system used in these materials is outstanding, reaching to the point of recreating rugosity from the 2D maps, the night-day cycle of the Earth and the atmosphere of Venus. The The material of the rings contains transparencies to faithfully recreate the many gaps in the Saturn rings, and a model of the rings was specifically created so that the materials and the mesh would coincide perfectly.

A total of seven case examples have been created based on seven different codes in order to both show the examples of what the toolkit is able to achieve and to better explain the functioning of the code and how to work with it. The results of the renders provided by these codes are remarkable, and they demonstrate the high degree of precision of the toolkit and its capabilities to help visualize the trajectories of spacecrafts or the observation by their instruments.

There are many ways the toolkit can be improved. New classes could be developed for other solar systems bodies apart from the planets, satellites and stars. Mainly, dwarf planets like Ceres or Eris, non-periodic and periodic comets like Halley, big asteroids or a model of the asteroid bell.

The treatment of the stars of the sky is also improvable. The current method used uses a fixed catalogue epoch, meaning that it is always the same position regardless of epoch or observer location. Although the relative error in the position of the stars can be negligible, another catalogue could be used that calculated the position of the stars at different epochs and locations.

The way point stars are represented can also be adjusted to better represent the starred night sky.

More materials could be created for the rest of solar system bodies following the same node pattern, as currently the lesser important satellites use a default generic material. There are some issues when rendering the rings of Saturn with Cycles compared to Eevee, because the their colour is darker and more brownish when reflected by the sun light, which seems to be caused by the transparencies of its texture.

More missions, such as the New Horizons or Messenger, could be added to the toolkit in order enlarge the pool of missions. This would require providing 3D models of their spacecrafts and updating the mission kernels and probe cameras data sets with the corresponding information. Future projects could also develop methods to help visualize the movement of the spacecrafts. For example, a colorful trail could be added to illustrate trajectory of spacecrafts or satellites, or a straight line pointing from the instruments to better their observations in the 3D view.

Acknowledgments

Special thanks to my family for providing me with invaluable support through the whole process. Special thanks too to both Manel Soria and Arnau Miró. Without their help this project wouldn't have arrived this far, and their assistance has been crucial in many aspects of the toolkit. Finally, thanks to all the people and organizations that provided this project with open-source and free resources.

References

[1] Spice Users. NAIF, 2020 [Consulted on April 29, 2020]. Available at: https://naif.jpl.nasa.gov/naif/SPICE_Users.pdf

[2] Three-dimensional representation of a spacecraft's trajectory. US5987363A US patent [Consulted on June 28, 2020]. Available at: https://patents.google.com/patent/US5987363A/en

[3] SPICE Module for the Satellite Orbit Analysis Program (SOAP). J. Coggi, R. Carnright, C. Hildebrand, [Consulted on June 28, 2020]. Available at: https://ntrs.nasa.gov/search.jsp?R=20080048138

[4] SPICE for ESA Planetary Missions: geometry and visualization support to studies, operations and data analysis. M. Costa, 2018 [Consulted on June 28, 2020]. Available at: https://meetingorganizer.copernicus.org/EPSC2018/EPSC2018-50.pdf

[5] How to Make Earth in Blender (Cycles). Blender Guru [Consulted on June 28, 2020]. Available at: https://youtu.be/9Q8PwcDzb8Y

[6] B3dplanetgen 1.0 Planet Generator. CD CARSWELL, 2018 [Consulted on June 28, 2020]. Available at: https://www.blendernation.com/2018/03/25/b3dplanetgen-1-0/

[7] WILL IT BLEND? GETTING SPICE-Y WITH DTMS AND PLANETARY VISUALIZATION. L. M. Davis, V. H. Silva, N. M. Estes, A.K. Boyd and K. S. Bowley, and the LROC Team, School of Earth and Space Exploration, Arizona State University, Tempe, AZ, 2017 [Consulted on June 28, 2020]. Available at: https://www.hou.usra.edu/meetings/planetdata2017/pdf/7073.pdf

[8] Videos by the LROC Team, From our Lunar Image Mosaics. LROC [Consulted on June 28, 2020]. Available at: http://lroc.sese.asu.edu/posts/videos

[9] IAU 2006 Resolution B3. IAU, 2006 [Consulted on May 6, 2020]. Available at: https://www.iau.org/static/resolutions/IAU2006_Resol3.pdf

[10] Transformations between Time Systems. Navipedia - GNSS Science Support Centre, 2011 [Consulted on May 6, 2020]. Available at: https://gssc.esa.int/navipedia/index.php/Transformations_between_Time_Systems

[11] Onboard Systems. NASA [Consulted on May 7, 2020]. Available at: https://solarsystem.nasa.gov/basics/chapter11-1

[12] Deep Space Atomic Clock (DSAC) Overview. NASA, 2019 [Consulted on May 7, 2020]. Available at: https://www.nasa.gov/mission_pages/tdm/clock/overview

[13] C. Ma, E. F. Arias, T. M. Eubanks, A. L. Fey, A.-M. Gontier, C. S. Jacobs, O. J. Sovers, B. A. Archinal, P. Charlot."The International Celestial Reference Frame as Realized by Very Long Baseline Interferometry". *The Astronomical Journal*, Vol. 116, No. 1, 516–546, (jul 1998). IOP Publishing. Available at: https://iopscience.iop.org/article/10.1086/300408/fulltext

[14] Fundamental Concepts. NAIF, 2020 [Consulted on May 6, 2020]. Available at: https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/Tutorials/pdf/individual_docs/04_concepts.pdf

[15] An Overview of Reference Frames and Coordinate Systems in the SPICE Context. NAIF, 2020 [Consulted on May 6, 2020]. Available at: https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/Tutorials/pdf/individual_docs/17_frames_and_coordinate_systems.pdf

[16] ATHENA - Coordinate System Document. EASA, 2015 [Consulted on May 6, 2020]. Available at: https://www.cosmos.esa.int/documents/400752/400864/ATHENA+-+Coordinate+System+Document+Issue+1.1.pdf/

[17] Rotation. NAIF, 2017 [Consulted on May 6, 2020]. Available at: https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/req/rotation

[18] Light Time Delay and Apparent Position. Analytical Graphics, Inc. [Consulted on May 8, 2020]. Available at: https://help.agi.com/stk/LinkedDocuments/LightTimeDelayandApparentPosition.pdf

[19] Charles Acton. Computing Space Mission Geometry Using NASA's SPICE System. NASA, 2019. [Consulted on April 28, 2020]. Available at: https://www.nasa.gov/smallsat-institute/computing-space-mission-geometry-using-nasa-spice-system

[20] The SPICE Concept. NAIF, 2020 [Consulted on April 29, 2020]. Available at: https://naif.jpl.nasa.gov/naif/spiceconcept

[21] About NAIF. NAIF, 2020 [Consulted on April 29, 2020]. Available at: https://naif.jpl.nasa.gov/naif/about

[22] Andrew Annex. SpiceyPy Documentation. [Consulted on May 7, 2020]. Available at: https://readthedocs.org/projects/spiceypy/downloads/pdf/latest

[23] Links to Related Sites. NAIF, 2020 [Consulted on May 2, 2020]. Available at: https://naif.jpl.nasa.gov/naif/links

[24] SPICE Kernel Required Reading. NAIF, 2014 [Consulted on April 29, 2020]. Available at: https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/req/kernel

[25] furnsh_c. NAIF, 2017 [Consulted on May 2, 2020]. Available at: https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/cspice/furnsh_c

[26] kinfo_c. NAIF, 2017 [Consulted on June 7, 2020]. Available at: https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/cspice/kinfo_c

[27] NAIF Integer ID codes. NAIF, 2017 [Consulted on May 3, 2020]. Available at: https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/req/naif_ids

[28] bodc2n_c. NAIF, 2017 [Consulted on May 3, 2020]. Available at: https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/cspice/bodc2n_c

[29] bodn2c_c. NAIF, 2017 [Consulted on May 3, 2020]. Available at: https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/cspice/bodn2c_c

[30] Time Routines in CSPICE. NAIF, 2015 [Consulted on April 29, 2020]. Available at: https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/req/time

[31] str2et_c. NAIF, 2017 [Consulted on May 2, 2020]. Available at: https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/cspice/str2et_c

[32] SCLK Required Reading. NAIF, 2010 [Consulted on April 29, 2020]. Available at: https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/req/sclk

[33] sce2c_c. NAIF, 2017 [Consulted on May 3, 2020]. Available at: https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/cspice/sce2c_c

[34] Reference Frames. NAIF, 2017 [Consulted on April 29, 2020]. Available at: https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/req/frames

[35] Cassini Spacecraft Frame Definitions Kernel. NAIF, 2008 [Consulted on May 7, 2020]. Available at: https://naif.jpl.nasa.gov/pub/naif/CASSINI/kernels/fk/cas_v40.tf

[36] Thomas A. Burk. "Cassini Orbit Trim Maneuvers at Saturn - Overview of Attitude Control Flight Operations". *American Institute of Aeronautics and Astronautics* [Consulted on May 10, 2020]. Available at: https://trs.jpl.nasa.gov/bitstream/handle/2014/43883/11-2675_A1b.pdf?sequence=1&isAllowed=y

[37] PCK Required Reading. NAIF, 2013 [Consulted on May 3, 2020]. Available at: https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/req/pck

[38] pxform_c. NAIF, 2017 [Consulted on May 3, 2020]. Available at: https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/cspice/pxform_c

[39] m2eul_c. NAIF, 2017 [Consulted on May 3, 2020]. Available at: https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/cspice/m2eul_c

[40] bodfnd_c. NAIF, 2017 [Consulted on May 3, 2020]. Available at: https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/cspice/bodfnd_c

[41] bodvrd_c. NAIF, 2017 [Consulted on May 3, 2020]. Available at: https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/cspice/bodvrd_c

[42] C-Kernel Required Reading. NAIF, 2014 [Consulted on May 3, 2020]. Available at: https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/req/ck

[43] ckgp_c. NAIF, 2017 [Consulted on May 3, 2020]. Available at: https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/cspice/ckgp_c

[44] SPK Required Reading. NAIF, 2017 [Consulted on May 4, 2020]. Available at: https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/req/spk

[45] Ephemeris Subsystem SPK. NAIF, 2020 [Consulted on May 10, 2020]. Available

at: https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/Tutorials/pdf/individual_docs/18_spk

[46] spkezr_c. NAIF, 2017 [Consulted on May 4, 2020]. Available at: https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/cspice/spkezr_c

[47] Introduction. Blender [Consulted on May 10, 2020]. Available at: https://docs.blender.org/manual/en/2.82/getting_started/about/introduction.html

[48] Blender Foundation. Blender [Consulted on May 10, 2020]. Available at: https://www.blender.org/foundation/

[49] Previous Versions. Blender [Consulted on May 10, 2020]. Available at: https://www.blender.org/download/previous-versions/

[50] Blender 2.82 Reference Manual. Blender [Consulted on May 12, 2020]. Available at: https://docs.blender.org/manual/en/2.82/index.html

[51] Introduction to to Files and Data System. Blender [Consulted on May 10, 2020]. Available at: https://docs.blender.org/manual/en/2.82/files/introduction.html

[52] Blender 2.82 Python API Documentation. Blender, 2020 [Consulted on May 13, 2020]. Available at: https://docs.blender.org/api/2.82/index.html

[53] Bernat Garreta, Arnau Miró, Manel Soria. mView (Mission View) Blender and SPICE toolkit to visualize SC missions. 2020. Available at: https://github.com/ArnauMiro/mView

[54] Cartography and Imaging Sciences Node. NASA. [Consulted on June 25, 2020]. Available at: https://pds-imaging.jpl.nasa.gov/

[55] IAU Catalog of Star Names (IAU-CSN). IAU Division C Working Group on Star Names (WGSN). [Consulted on June 23, 2020]. Available at: http://www.pas.rochester.edu/~emamajek/WGSN/IAU-CSN.txt

[56] Naming stars. International Astronomical Union. [Consulted on June 23, 2020]. Available at: https://www.iau.org/public/themes/naming_stars/

[57] Earth Rising over the Moon's Horizon. NASA, 2019. [Consulted on June 23, 2020] Available at: https://www.nasa.gov/image-feature/earth-rising-over-the-moons-horizon

[58] Recreation of Inner Solar System Planets. Bernat Garreta. Available at: https://youtu.be/3u2zUK4QluQ

[59] Stellarium Astronomy Software. Stellarium. [Consulted on June 27, 2020] Available at: https://stellarium.org/

[60] Space Imeages | Bright Moons. NASA [Consulted on June 27, 2020] Available at: https://www.jpl.nasa.gov/spaceimages/details.php?id=PIA12771

[61] Pan In the Middle | NASA Solar System Exploration. NASA [Consulted on June 27, 2020]

Available at: https://solarsystem.nasa.gov/resources/15638/pan-in-the-middle/

[62] Recreation of Cassini Insertion into Saturn orbit. Bernat Garreta. Available at: https://youtu.be/sqaFySPHu-U

[63] Space Imeages | Volcanic Eruptions on Io. NASA [Consulted on June 28, 2020] Available at: https://www.jpl.nasa.gov/spaceimages/details.php?id=PIA00379

[64] Requirements. Blender [Consulted on June 28, 2020]. Available at: https://www.blender.org/download/requirements/